Programming XC on XMOS Devices

Douglas Watt

Updated versions of this document are available at:
XC Programming Guide: http://www.xmos.com/xc-programming-guide
XC Specification: http://www.xmos.com/xc-specification

XMOS™

Programming XC on XMOS Devices
by Douglas Watt

XMOS™

The authors typeset this book using LaTeX in Lucida Bright, Lucida Sans and Computer Modern Typewriter. Tom Hunt produced the XMOS documentation build system.

XMOS also publishes its books in electronic formats. Some content that appears in print may not be available in electronic books.

For information on XMOS products, visit us on the Web: `www.xmos.com`.

Because of the dynamic nature of the Internet, any Web addresses or links contained in this book may have changed since publication and may no longer be valid.

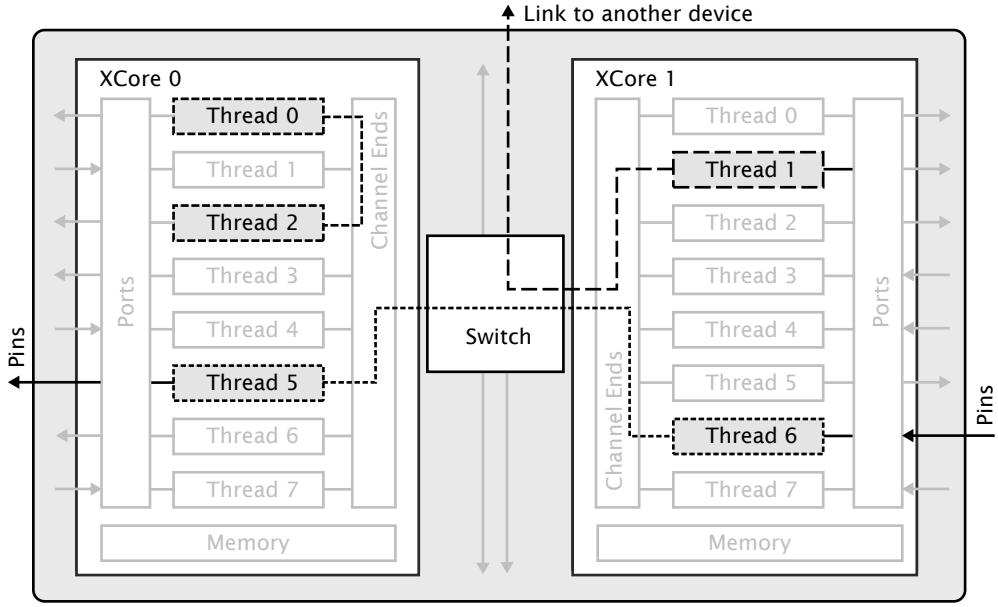
Printed and bound by CPI Antony Rowe, Chippenham.

ISBN: 978-1-907361-00-5 (PBK)
ISBN: 978-1-907361-03-6

Published by XMOS Limited.

Welcome to XMOS

The XMOS architecture enables a combination of interface, digital signal processing and control functions to be performed in software. An XMOS device consists of one or more XCores, each comprising an event-driven multi-threaded processor architected for real-time performance with tightly integrated I/O and on-chip memory. Each processor has hardware support for executing several threads concurrently and has dedicated instructions for input and output.



The architecture is deterministic, with each thread guaranteed a slice of the processing. The threads can execute computations, handle real-time I/O operations and respond to multiple events. The I/O pins can be sampled or driven using a single instruction, and data rates can be controlled using timers or clocks. A high-performance switch enables communication between processors and makes it easy

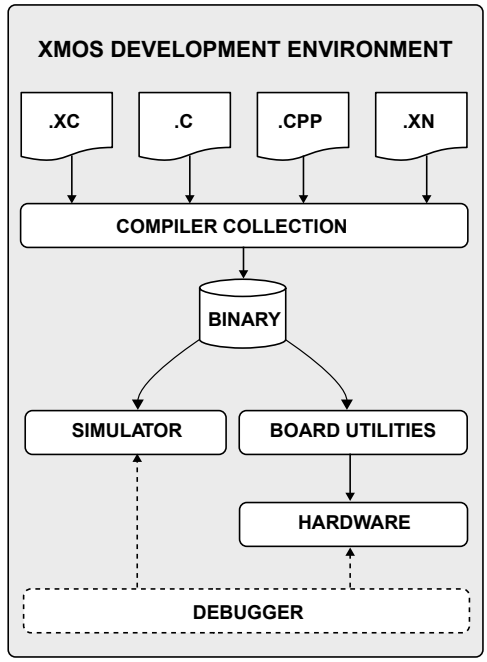
to construct systems from multiple devices. Communication between threads on a processor incurs no latency, and between processors the latency can be determined for a known communication pattern. The hardware is described in a separate book [1].

Programming Model

Programs are written using a combination of XC, C and C++. XC provides extensions to C that simplify the control over concurrency, I/O and time. These extensions map directly onto XCore hardware resources such as threads, channels and ports, avoiding the need to make extensive use of library calls. XC's constructs are *efficient*—compiling into short instruction sequences, and *safe*—free from many sources of deadlock, race conditions and memory violations. This makes programs easy to write, understand and debug. The XC language is fully described in this book.

Tools Architecture

The XMOS tools are based on a standard embedded development flow and are built upon industry-standard platforms, making them intuitive and easy to use.



The concurrent and real-time capabilities of the language and architecture are supported by the tools through all stages of development:

- The compiler toolchain statically analyses generated binary code, attempting to prove timing assertions specified in the source. This results in code that is portable across multiple devices with different timing characteristics.
- The compiler toolchain generates a single binary file that includes instruction and data segments for all devices. The simulator and board utilities operate on this file format, hiding complexity from the programmer.
- An XN file describes the target platform, which may include networks of XMOS devices, SPI flash memory, an oscillator and JTAG scan chain. XN files allow the tools to fully automate system boot and configuration.
- The board utilities can configure a system to boot from a host development PC, on-board flash memory or on-chip OTP memory.
- The debugger interacts with all processors on the target platform, presenting a collection of threads to the programmer that can be viewed together. This allows XMOS devices to be debugged in the same way as conventional processors.

The tools are described in a separate user guide [2].

How to Read This Book

The main chapters in this book form a tutorial on how to write XC programs for the XMOS architecture. The chapters are complemented by a set of appendices that provide the full XC specification and details of the XC implementation on the first generation of XMOS devices: the XS1. The tutorial assumes some prior programming experience.

Chapter 1 outlines the support for computation in XC, including arithmetic expressions, control-flow constructs and functions.

Chapter 2 explains how to input and output data on ports, which interface a device with external components. It shows how to control I/O data rates using timers and how to interface multiple components using the `select` statement.

Chapter 3 shows how to run multiple tasks concurrently as separate threads that use channels to communicate with one another.

Chapter 4 describes how to synchronise I/O operations to a clock, recording and controlling on which edge each input and output occurs.

Chapter 5 demonstrates how buffers in the ports can improve overall performance by decoupling I/O operations from computations.

Chapter 6 explains how to serialise data onto I/O pins and how to interpret and generate strobe signals using built-in capabilities of the ports.

Appendix A provides the official XC language specification.

Appendix B provides the official XC I/O semantics of ports.

Appendix C documents the XS1 implementation of XC, including support for the different I/O operations, the port-to-pin mapping, and the size and alignment of XC's data types.

The computational framework of XC is similar to C, in particular with respect to its type system and control-flow constructs. Experienced C programmers may therefore wish to read Chapters 2–6 first, which cover the I/O and concurrency constructs. These chapters should be read in order as each chapter builds upon concepts introduced in previous chapters.

Development of the XMOS architecture and XC language are continuing. The current implementation is discussed in Appendix C, which also serves as a roadmap for future direction and standardisation of the technology.

Acknowledgements

Many of the examples of port usage were originally developed by Henk Muller for the XS1 ports tutorial [3]. The case studies which develop an LCD driver and Ethernet controller were developed jointly with Larry Snizek. Richard Osborne wrote the XS1 standard library documentation and Huw Geddes painstakingly produced all of the waveform diagrams and illustrations.

Ali Dixon, Peter Hedinger, Russell Gallop and many others have carefully reviewed drafts of the English manuscript. Their comments, corrections and suggestions have significantly improved the quality of the final text.

Contents

CHAPTER 1

Computation

XC is an imperative programming language with a computational framework based on C. XC programs consist of functions that execute statements that act upon values stored in variables. Control-flow statements express decisions, and looping statements express iteration.

In the following sections, constructs that are new to XC or that differ from C are noted in the margin.

NEW XC

1.1 Hello, World!

The first task often performed when learning a new programming language is to print the words "Hello, world!" A suitable XC program is shown below.

```
#include <stdio.h>

main(void) {
  printf("Hello, world!\n");
}
```

The first line of this program tells the compiler to include information from the header file `stdio.h`. This file contains a declaration of the function `printf`, which outputs a string to standard output, for example a terminal window on a development system.

Every program must contain a single `main` function, which is where the program begins executing. In this example, `main` is defined as a function that expects no arguments, indicated by the keyword `void`.

The body of a function is enclosed in braces `{}` and contains statements that specify operations to be performed. In this example, `main` contains a single statement

that calls the function `puts` with a string literal as its argument. The escape sequence `\n` denotes a newline character.

1.2 Variables, Constants and Expressions

A variable represents a location in memory in which data is storred. All variables must be declared before use and given a type. The most common arithmetic types are `char` and `int`. A `char` is a byte that represents 8-bit integral numbers and an `int` represents 32-bit integral numbers. The declaration

```
char c;
```

declares `c` to be an 8-bit signed character that takes values between -128 and 127.

The qualifier `signed` or `unsigned` may be used to specify the signedness of a type. The declaration

```
unsigned char c;
```

declares `c` to be an 8-bit unsigned character that takes values between 0 and 255.

A variable may be assigned an initial value. The declaration

```
int i = 0, j = 1;
```

declares `i` and `j` to be integers, initialised with the values 0 and 1.

The qualifier `const` may be applied to any variable declaration to prevent its value from being changed after initialisation. The declaration

```
const int MHz = 1000000;
```

declares `MHz` to represent an integer constant of value 1000000. Attempting to modify its value after initialisation is invalid.

One or more variables of the same type may be combined to form an *array*. The declaration

```
int data[3] = {1, 2, 3};
```

declares `data` to be an array of three integers and initialises it with values 1, 2 and 3. Array subscripts start at zero, so the elements of this array are `data[0]`, `data[1]` and `data[2]`. A subscript can be any integer expression that evaluates to a valid element of the array.

Arrays may be constructed from one another to form *multi-dimensional* arrays. The declaration

```
int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

declares `matrix` to be a two-dimensional array. The first dimension specifies a row, the second a column, producing the matrix below.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

The subscripts are ordered by the largest dimension first so that, for example, the value of `matrix[0][1]` is 2.

1.2.1 Constants

A constant is a textual representation of a value, and has a data type. Entries in the table below are all examples of constants.

Text	Type	Value
`123`	`int`	123
`123u`	`unsigned int`	123
`0b10000`	`int`	16
`020`	`int`	16
`0x10`	`int`	16
`0xFu`	`unsigned int`	10
`'x'`	`char`	120
`'0'`	`char`	48
`'\n'`	`char`	10 (newline)
`'\\'`	`char`	92 (backslash)
`'\0'`	`char`	0 (null terminator)
`"str"`	array of `char`	`'s', 't', 'r', '\0'`

A sequence of digits is by default an `int`. An unsigned constant is specified with the suffix `u`. An integer constant is specified in binary by using the prefix `0b`, in octal by using the prefix `0` and in hexadecimal by using the prefix `0x`.

A character constant is usually written as a character in single quotes. Its value is the numeric value of the character. Some characters that are not representable as characters are *escaped* using the backslash character.

A *string literal* is a sequence of zero or more characters enclosed in double quotes. The internal representation of a string literal includes a null character suffix `\0`, which allows programs to find the end of a string. This suffix also increases the string's storage requirements by a single byte. String literals are used to initialise arrays of characters, as in:

```
char msg[] = "Hello, world!\n";
```

This example declares `msg` to be an array of 15 characters, including the null terminator. If the size of the array is specified in the declaration, it must be at least as large as the string.

1.2.2 Expressions

An expression combines variables and constants with operators, producing a value. Entries in the table below are all examples of expressions.

Algebraic Expression	XC Expression
$a * b - c$	`a * b - c`
$(a + b)(c + d)$	`(a + b) * (c + d)`
$a/b + c$	`(a/b) + c`

An expression without parenthesis is usually evaluated from left to right using the rules of precedence of operators. These rules state that the * operator has a higher precedence than the + operator, which means that the second expression in the table requires parenthesis around the two additions to force the required grouping.

Table I summarises the expression operators supported in XC. Operators higher in the table have a higher precedence, and operators in the same section have the same precedence. The operators are defined to have the same meaning as in C; full details are given in §A.6.

An expression becomes a statement when followed by a semicolon. Most statements are either assignments, as in:

```
x = a * b;
```

or function calls, as in:

```
printf("Hello, world!\n");
```

NEW XC

The value of an expression must be unambiguous. An ambiguity arises if the value of an expression depends on the order of evaluation of its operands, as in:

```
i = i++;  /* invalid */
```

In this example, the value of `i` depends on the order in which the assignment and increment operators are performed.

In general, if one subexpression contains a modification of variable V, none of the other subexpressions are allowed to use V. This rule applies recursively to functions called in expressions that read or write global variables.

1.2.3 Type Conversions

If an operator has operands of different types, the operands are converted to a common type. In general, the "lower" type is *promoted* to the "higher" type before the operation proceeds; the result is of the higher type. For example, in the expression

```
'c' + 1
```

the binary operator + takes a `char` and an `int` operand. The `char` operand is promoted to an `int`, and the result of the expression is an `int`.

TABLE I
XC EXPRESSION OPERATORS

Operator	Description	Type	Associativity
`++ --`	Postfix increment/decrement	Unary	left-to-right

`++ --`
`+ -`
`!`
`~`
(*type*)
`sizeof`
`sizeof`
	Prefix increment/decrement
	Unary plus/minus
	Logical negation
	Bitwise complement
	Explicit cast
	Determine size in bytes
	Determine whether null reference
		Unary	right-to-left

`* / %`	Multiplication/division/modulus	Binary	left-to-right
`+ -`	Addition/subtraction	Binary	left-to-right
`<< >>`	Bitwise shift left/right	Binary	left-to-right

`<`
`<=`
`>`
`>=`
	Relational less than
	Relational less than or equal to
	Relational greater than
	Relational greater than or equal to
		Binary	left-to-right

`== !=`	Relational equal to, not equal to	Binary	left-to-right
`&`	Bitwise AND	Binary	left-to-right
`^`	Bitwise exclusive OR	Binary	left-to-right
`|`	Bitwise inclusive OR	Binary	left-to-right
`&&`	Logical AND	Binary	left-to-right
`||`	Logical OR	Binary	left-to-right
`c?`t`:`f	Ternary conditional	Ternary	right-to-left

`=`
`+= -= *= /=`
`%= &= ^= |=`
`<<= >>=`
	Assignment
	Arithmetic assignment
	Arithmetic assignment
	Arithmetic assignment
		Binary	right-to-left

The general rules of promotion and arithmetic conversion are stated in §A.1.1, and for XS1 devices can be summarised as follows:

- Convert `char` and `short` to `int`, if an `int` can represent all the values of the original type, otherwise convert to `unsigned int`.
- If either operand is `unsigned int`, convert the other to `unsigned int`.

Explicit type conversions can be forced in an expression using the unary cast operator, as in:

```
(char)('a' + i); // cast 32-bit integer to an 8-bit char
```

Casts are often used with output statements to specify the amount of data to be communicated (see §3.3). The meaning of the cast is as if the expression were assigned to a variable of the specified type. A cast must not specify an array; neither must the expression.

1.3 Control Flow

Control-flow statements express decisions that determine the order in which statements are performed. A list of statements is executed in sequence by grouping them into a *block* using braces `{ }`, as in:

```
main(void) {
  int x = 2, y = 3;
  int z = x * y;
  z++;
}
```

In a block, all declarations must come at the top before the statements. A block is syntactically equivalent to a single statement and can be used wherever a statement is required.

1.3.1 If-Else

An `if-else` construct choses at most one statement to execute, as in:

```
if (n > 0)
  printf("Positive");
else if (n < 0) {
  printf("Negative");
  x = 0;
}
else
  printf("Zero");
```

Each parenthesised expression guards a statement or block that may be executed; the `else-if` and `else` statements are optional. The expressions are evaluated in the order they appear in the source code, and the first expression that produces a non-zero value causes the statement it guards to be executed. The entire construct then terminates.

An `else` statement is always associated with the previous `else`-less `if`, which is important to remember if multiple `if` statements are nested. Braces can be used to force a different association.

1.3.2 Switch

A `switch` statement tests whether an expression matches one of a number of constant integer values and branches to the body of code for the corresponding `case`, as in:

```
switch(state) {
  case READY :
    state = SET;
    if(x < 0)
      state = FAIL;
    break;
  case SET :
     state = GO;
     if(y > 0)
       state = FAIL;
     break;
  case GO:
    printf("Go!\n");
    break;
  case FAIL :
  default :
    /* error */
}
```

If a `default` label is present and none of the case constants equal the value of the expression, the code following `default` is executed instead.

The body of each `case` must be terminated by either `break` or `return`, preventing control from flowing from one body to the next.

NEW XC

1.3.3 Loops

A `while` loop repeats a statement as long as the value of an expression remains non-zero, as in:

```
int i = 0;
while (i<n) {
  a[i] = b[i] * c[i];
  i++;
}
```

This example is typical of many programs that iterate over the first *n* elements of an array. An alternative form is to use a `for` loop, which provides a way to combine the initialisation, conditional test and increment together at the top of the loop. The example above may be alternatively written as:

```
for (int i=0; i<n; i++)
  a[i] = b[i] * c[i];
```

The first and third expressions are usually assignments, and the second a relational expression. The first assignment may form part of a variable declaration whose scope is local to the body of the loop. Any of these three parts may be omitted, but the semicolons must remain.

A `do-while` loop performs the test after executing its body, guaranteeing that its body is executed at least once. Its form is shown below.

```
do {
  body
} while (exp);
```

1.3.4 Break and Continue

A `break` statement exits from a loop immediately, as in:

```
while (1) {
  //...input and process data...
  if (error)
    break;
}
```

In this example, the `break` statement exits from the `while` loop upon encountering an error. In general, `break` causes the innermost enclosing loop or `switch` statement to exit.

A `continue` statement is similar to `break`, except that it causes the next iteration of the enclosing loop to begin, as in:

```
for (int i=0; i<n; i++) {
  if (a[i] == 0)
    continue;
  //...process non-zero elements...
}
```

In a `for` loop, the statement executed immediately after `continue` is the loop increment. In `while` and `do` loops, the next statement executed is the conditional test.

A `continue` statement is often used where the code that follows it is complicated, so that reversing the test and indenting another level would nest the program too deeply to be easily understood.

1.4 Functions

A function names a block of statements, providing a way to encapsulate and parameterise a computation. The program below defines and uses a function `fact`, which computes a factorial.

```
int fact(int);

/* test fact function */
int main(void) {
  for (int i=0; i<10; i++) {
    int f = fact(i);
    // ... print f ...
  }
}

int fact(int n) {
  for (int i=n-1; i>1; i--)
    n = n * i;
  return n;
}
```

The declaration of `fact` after `main`

```
int fact(int n)
```

declares `fact` to be a function that takes an `int` parameter named `n` and returns an `int` value. When `main` calls `fact`, the value of `i` is copied into a new variable `n`. The variable `n` is private to `fact`, and other functions can use this name without conflict.

The block of statements grouped in braces `{ }` following the declaration of `fact` makes it a *definition*. At most one definition of each function is permitted.

The `return` statement in the body of `fact` returns the computed factorial value to `main`. A function that does not return a value is specified with the return type `void`.

The first declaration of `fact` before `main`

```
int fact(int);
```

is a *prototype* that declares the type of `fact` without giving it a definition. Either a function prototype or its definition must appear in the source code before the function is used. The protototpe must agree with its definition and all of its uses; the parameter names in the prototype are optional.

1.4.1 Function Arguments

Arguments to functions are usually passed by *value*, in which case the value is copied into a new variable that is private to the function.

NEW XC

An argument may also be passed by *reference* so that any change made to the local variable also modifies the argument in the calling function. The program below swaps the values of two variables passed by reference.

```
void swap(int &x, int &y) {
  int tmp = x;
  x = y;
  y = x;
}

int main(void) {
  int a = 1;
  int b = 2;
  swap(a, b);
}
```

The declaration

```
void swap(int &x, int &y)
```

declares `swap` to be a function that accepts two variables by reference. A reference parameter is specified by prefixing its name with `&`.

NEW XC

The creation of more than one reference to the same object is invalid. In the above example, calling the function `swap` using the same variable twice would be invalid.

Arrays are implicitly passed by reference. This means that an array cannot be passed to two parameters of a function but, for example, passing two different rows of a two-dimensional array to a function is permitted.

The largest dimension of an array parameter may be omitted from the function declaration, allowing the function to operate on arbitrary sized arrays, as in:

```
int strcount(char str[], int len);
```

If the size of an array parameter is specificed, passing an array of larger size is permitted but the highest elements are not accessible; passing a smaller array is invalid.

1.4.2 Optional Arguments

NEW XC

A pass-by-reference parameter can be specified *nullable*, meaning that it can contain either a valid reference or a special `null` reference. The program below determines how many corresponding elements of two arrays have the same value, assigning the value 0 or 1 to the element of a third array only if provided by the caller.

```
int compare(int x[], int y[], int ?matches[], unsigned size) {
  int n = 0;
  for (int i=0; i<size; i++) {
    int match = (x[i] == y[i]);
    n += match;
    if (!isnull(matches))
      matches[i] = match;
  }
  return n;
}
```

```
int main(void) {
  int x[5] = {0, 1, 2, 3, 4};
  int y[5] = {1, 1, 1, 3, 3};
  int z[5] = {1, 1, 1, 1, 1};
  int m[5] = {0};
  int n;

  (void)compare(x, y, m, 5);
  n = compare(y, z, null, 5);

  return 0;
}
```

The declaration

```
int compare(int x[], int y[], int ?m[], unsigned size)
```

declares `compare` to be a function that accept three arrays and a `size` variable. The third parameter `m` is specified as nullable by prefixing its name with `?`.

The operator `isnull` produces a value 1 if its argument is a valid reference and 0 otherwise. Attempting to dereference or use a null object is invalid.

On the first call by `main` to `compare`, the array `matches` is passed as the third argument; `compare` assigns the elements of this array. On the second call, `null` is passed as the third argument; `compare` does not attempt to assign to the array.

1.4.3 Multiple-Return Functions

NEW XC

A function may be declared as returning more than one value, as in:

```
{int, int} swap(int a, int b) {
  return {b, a};
}

void main(void) {
  int a  = 1;
  int b  = 2;
  {a, b} = swap(b, a);
}
```

The list of return types, the list of values following `return`, and the list of variables assigned are enclosed in braces. The number of elements in the assignment list must match the number of values returned by the function, but any of the returned values may be ignored using `void`, as in:

```
{a, void} = f();
```

1.5 Reinterpretation

NEW XC

A reinterpretation causes a variable to be treated as having a different type, but it undergoes no conversion. The function below uses a reinterpretation to transmit an array of bytes as 32-bit integers.

```
void transmitMsg(char msg[], int nwords) {
  for (int i=0; i<nwords; i++)
    transmitInt((msg, int[])[i]);
}
```

The construction

```
(msg, int[])
```

reinterprets the array `msg` as an array of integers, which is then indexed, as in:

```
(msg, int[])[i]
```

In this example, the size of the integer array is determined at run-time. If the function is called, for example, with an array of 10 bytes, the reinterpreted integer array has an upper bound of 2 and the topmost 2 characters are inaccessible in the reinterpretation. If size of the reinterpretation is given, it must not exceed the size of the original type. Attempting to reinterpret one object to another whose type requires greater storage alignment (as specified in §C.5) is invalid. The original declaration should specify the largest storage alignment required for all possible reinterpretations.

1.6 Comparison with C

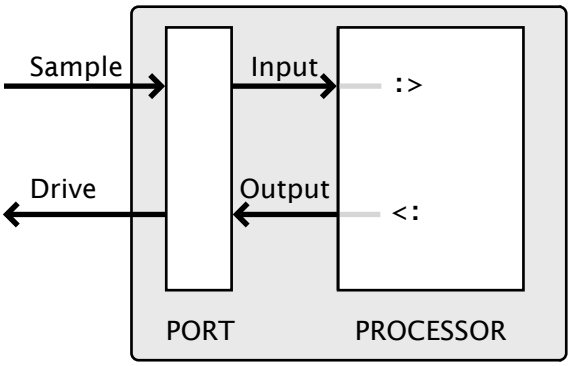
XC provides many of the same capabilities as C, the main omission being support for pointers. Consequently, many programming errors that are undefined in C are known to be invalid in XC and can be caught either by the compiler or raised as run-time exceptions. All of XC's data types and operators have the same meaning as in C, and user-defined types including structures, unions, enumerations and typedefs are also supported. The extensions for pass-by-reference parameters and multiple-return functions provide support for operations usually performed using pointers in C. XC's scope and linkage rules are the same as with C, and both languages use the same preprocessor.

XC does not support floating point, `long long` arithmetic, structure bit-fields or volatile data types, and no `goto` statement is provided. These restrictions may be relaxed in future releases to improve compatibility between languages.

CHAPTER 2

Input and Output

A port connects a processor to one or more physical pins and as such defines the interface between a processor and its environment. The port logic can drive its pins high or low, or it can sample the value on its pins, optionally waiting for a particular condition. Ports are not memory mapped; instead they are accessed using dedicated instructions. XC provides integrated input and output statements that make it easy to express operations on ports. The diagram below illustrates these operations.



Data rates can be controlled using hardware timers that delay the execution of the input and output instructions for a defined period. The processor can also be made to wait for an input from more than one port, enabling multiple I/O devices to be interfaced concurrently.

2.1 Outputting Data

A simple program that toggles a pin high and low is shown below.

```
#include <xs1.h>

out port p = XS1_PORT_1A;

int main(void) {
  p <: 1;
  p <: 0;
}
```

The declaration

```
out port p = XS1_PORT_1A;
```

declares an output port named `p`, which refers to the 1-bit port identifier 1A.[1]

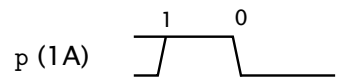
The statement

```
p <: 1;
```

outputs the value 1 to the port `p`, causing the port to drive its corresponding pin high. The port continues to drive its pin high until execution of the next statement

```
p <: 0;
```

which outputs the value 0 to the port, causing the port to drive its pin low. The diagram below shows the output generated by this program.



The pin is initially not driven; after the first output is executed it is driven high; and after the second output is executed it is driven low. In general, when outputting to an *n*-bit port, the least significant *n* bits of the output value are driven on the pins and the rest are ignored.

All ports must be declared as global variables, and no two ports may be initialised with the same port identifier. After initialisation, a port may not be assigned to. Passing a port to a function is allowed as long as the port does not appear in more than one of a function's arguments, which would create an illegal alias.

[1] The value `XS1_PORT_1A` is defined in the header file `<xs1.h>`. Most development boards are supplied with an XN file from which the header file `<platform.h>` is generated, and which defines more intuitive names for ports such as `PORT_UART_TX` and `PORT_LED_A`. These names are documented in the corresponding hardware manual.

2.2 Inputting Data

The program below continuously samples the 4 pins of an input port, driving an output port high whenever the sampled value exceeds 9.

```
#include <xs1.h>

in  port  inP = XS1_PORT_4A;
out port outP = XS1_PORT_1A;

int main(void) {
  int x;
  while (1) {
    inP :> x;
    if (x > 9)
      outP <: 1;
    else
      outP <: 0;
  }
}
```

The declaration

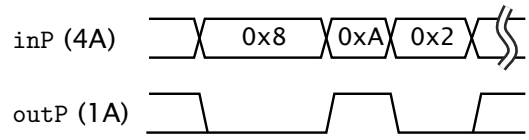
```
in port inP = XS1_PORT_4A;
```

declares an input port named `inP`, which refers to the 4-bit port identifier 4A.

The statement

```
inP :> x;
```

inputs the value sampled by the port `inP` into the variable `x`. The diagram below shows example input stimuli and expected output for this program.



The program continuously inputs from the port `inP`: when 0x8 is sampled the output is driven low, when 0xA is sampled the output is driven high and when 0x2 is sampled the output is again driven low. Each input value may be sampled many times.

2.3 Waiting for a Condition on an Input Pin

An input operation can be made to wait for one of two conditions on a pin: equal to or not equal to some value. The program on the next page uses a *conditional input* to count the number of transitions on its input pin.

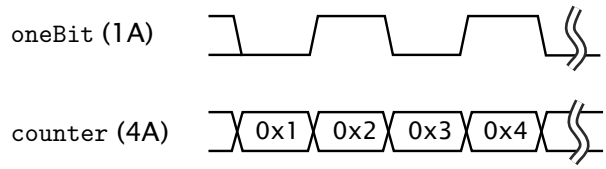
```
#include <xs1.h>

in  port oneBit  = XS1_PORT_1A;
out port counter = XS1_PORT_4A;

int main(void) {
  int x;
  int i = 0;

  oneBit :> x;
  while (1) {
    oneBit when pinsneq(x) :> x;
    counter <: ++i;
  }
}
```

The statement

```
oneBit when pinsneq(x) :> x;
```

instructs the port `oneBit` to wait until the value on its pins is not equal to `x` before sampling and providing it to the processor to store in `x`.

The waveform diagram below shows example input stimuli and expected output for this program.



As another example, the only operation required to wait for an Ethernet preamble on a 4-bit port is

```
ethData when pinseq(0xD) :> void;
```

The processor must complete an input operation from the port once a condition is met, even if the input value is not required. This is expressed in XC as an input to `void`.

Using a conditional input is more power efficient than polling the port in software, because it allows the processor to idle, consuming less power, while the port remains active monitoring its pins.

2.4 Controlling I/O Data Rates with Timers

A timer is a special type of port used for measuring and controlling the time between events. A timer has a 32-bit counter that is continually incremented at a rate of

100MHz and whose value can be input at any time. An input on a timer can also be delayed until a time in the future. The program below uses a timer to control the rate at which a 1-bit port is toggled.

```
#include <xs1.h>
#define DELAY 50000000

out port p = XS1_PORT_1A;

int main(void) {
  unsigned state = 1, time;
  timer t;
  t :> time;
  while (1) {
    p <: state;
    time += DELAY;
    t when timerafter(time) :> void;
    state = !state;
  }
}
```

The declaration

```
timer t;
```

declares a timer named `t`, obtaining a timer resource from the XCore's pool of available timers.

The statement

```
t :> time;
```

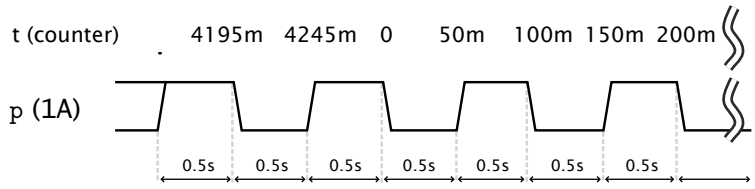
inputs the value of `t`'s counter into the variable `time`. This variable is then incremented by the value `DELAY`, which specifies a number of counter increments. The timer has a period of 10ns, giving a time in the future of $50,000,000 * 10ns = 0.5s$.

The conditional input statement

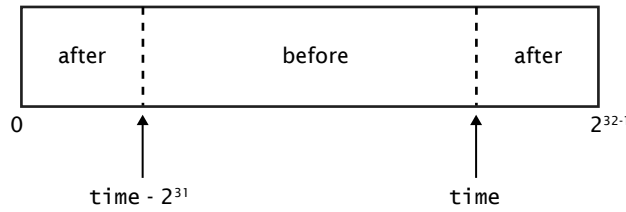
```
t when timerafter(time) :> void;
```

waits until this time is reached, completing the input just afterwards.

The waveform diagram below shows the data driven for this program.



The function `timerafter` treats the timer's counter as having two separate ranges, as illustrated below.



All values in the range (`time` -2^{31}..`time` -1) are considered to come before `time`, with values in the range (`time` $+1$..`time` $+2^{32-1}$, 0..`time` -2^{31}) considered to come afterwards. If the delay between the two input values fits in 31 bits, `timerafter` is guaranteed to behave correctly, otherwise it may behave incorrectly due to overlow or underflow. This means that a timer can be used to measure up to a total of $2^{31}/100,000,000 = 21s$.

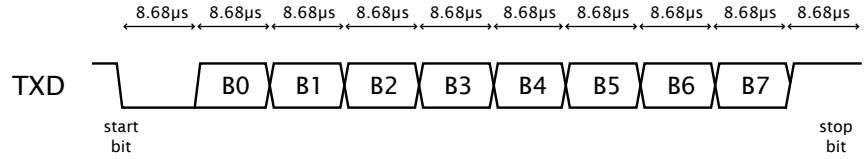
A programming error may be introduced by inputting the new time instead of ignoring it with a cast to `void`, as in

```
t when timerafter(time) :> time;
```

Because the processor completes the input shortly after the time specified is reached, this operation actually increments the value of `time` by a small additional amount. This amount may be compounded over multiple loop iterations, leading to signal drift and ultimately a loss of synchronisation with a receiver.

2.5 Case Study: UART (Part 1)

A universal asynchronous receiver/transmitter (UART) translates data between parallel and serial forms for communication over two 1-bit wires at fixed data rates. Each bit of data is driven for the time defined by the data rate, and the receiver must sample the data during this time. The diagram below shows the transmission of a single byte of data at a rate of 115200 bits/s.



The quiescent state of the wire is high. A byte is sent by first driving a *start bit* (0), followed by the eight data bits and finally a *stop bit* (1). A rate of 115200 bits/s means that each bit is driven for $\frac{1}{115200} = 8.68\mu s$.

UARTs are often implemented with microcontrollers, using interrupts to schedule memory-mapped input and output operations. Implementing a UART with an XMOS

device is easy due to its dedicated I/O instructions. The program below defines a UART transmitter.

```
#include <xs1.h>

#define BIT_RATE 115200
#define BIT_TIME 100000000 / BIT_RATE

out port TXD = XS1_PORT_1A;
out port RXD = XS1_PORT_1B;

void transmitter(out port TXD) {
  unsigned byte, time;
  timer t;

  while (1) {
    /* get next byte to transmit */
    byte = getByte();
    t :> time;

    /* output start bit */
    TXD <: 0;
    time += BIT_TIME;
    t when timerafter(time) :> void;

    /* output data bits */
    for (int i=0; i<8; i++) {
      TXD <: >> byte;
      time += BIT_TIME;
      t when timerafter(time) :> void;
    }

    /* output stop bit */
    TXD <: 1;
    time += BIT_TIME;
    t when timerafter(time) :> void;
  }
}
```

The transmitter outputs a byte by first outputting a start bit, followed by a conditional input on a timer that waits for the bit time to elapse; the data bits and stop bit are output in the same way.

The output statement in the `for` loop

```
TXD <: >> byte;
```

includes the modifier >>, which right-shifts the value of `byte` by the port width (1 bit) after outputting the least significant port-width bits. This operation is performed in the same instruction as the output, making it more efficient than performing the shift as a separate operation afterwards.

The function below receives a stream of bytes over a 1-bit wire.

```
void receiver(in port RXD) {
  unsigned byte, time;
  timer t;

  while (1) {
    /* wait for start bit */
    RXD when pinseq(0) :> void;
    t :> time;
    time += BIT_TIME/2;

    /* input data bits */
    for (int i=0; i<8; i++) {
      time += BIT_TIME;
      t when timerafter(time) :> void;
      RXD :> >> byte;
    }

    /* input stop bit */
    time += BIT_TIME;
    t when timerafter(time) :> void;
    RXD :> void;

    putByte(byte >> 24);
  }
}
```

The receiver samples the incoming signal, waiting for a start bit. After receiving this bit, it waits for $1\frac{1}{2}$ times the bit time and then samples the wire at the midpoint of the the first byte transmission, with subsequent bits being sampled at 8.68µ*s* increments. The input statement in the `for` loop

```
RXD :> >> byte;
```

includes the modifier >>, which first right-shifts the value of `byte` by the port width (1 bit) and then inputs the next sample into its most significant port-width bits. The expression in the final statement

```
putByte(byte >> 24);
```

right-shifts the bits in the integer `byte` by 24 bits so that the input value ends up in its least significant bits.

2.6 Responding to Multiple Inputs

The program below inputs two streams of data from two separate ports using only a single thread. The availability of data on one of these ports is signalled by the toggling of a pin, with data on another other port being received at a fixed rate.

```
#include <xs1.h>

#define DELAY_Q 10000000

in port toggleP = XS1_PORT_1A;
in port dataP   = XS1_PORT_4A;
in port dataQ   = XS1_PORT_4B;

int main(void) {
  timer t;
  unsigned time, x = 0;

  t :> time;
  time += DELAY_Q;
  while (1)
    select {
      case toggleP when pinsneq(x) :> x :
        readData(dataP);
        break;
      case t when timerafter(time) :> void :
        readData(dataQ);
        time += DELAY_Q;
        break;
    }
}
```

The `select` statement performs an input on either the port `toggleP` or the timer `t`, depending on which of these resources becomes ready to input first. If both inputs become ready at the same time, only one is selected, the other remaining ready on the next iteration of the loop. After performing an input, the body of code below it is executed. Each body must be terminated by either a `break` or `return` statement.

Case statements are not permitted to contain output operations as the XMOS architecture requires an output operation to complete but allows an input operation to wait until it sees a matching output before committing to its completion.

Each port and timer may appear in only one of the `case` statements. This is because the XMOS architecture restricts each port and timer resource to waiting for just one condition at a time.

In this example, the processor effectively multi-tasks the running of two independent tasks, and it must be fast enough to process both streams of data in real-time. If this is not possible, two separate threads may be used to process the data instead (see Chapter 3).

2.7 Case Study: UART (Part 2)

The program on the following page uses a `select` statement to implement both the transmit and receive sides of a UART in a single thread.

```
void UART(port RX, int rxPeriod, port TX, int txPeriod) {
  int txByte, rxByte;
  int txI, rxI;
  int rxTime, txTime;
  int isTX = 0;
  int isRX = 0;
  timer tmrTX, tmrRX;
  while (1) {
    if (!isTX && isData()) {
      isTX = 1;
      txI = 0;
      txByte = getByte();
      TX  <: 0;             // transmit start bit
      tmrTX :> txTime;      // set timeout for data bit
      txTime += txPeriod;
    }
    select {
      case !isRX => RX when pinseq(0) :> void :
        isRX = 1;
        tmrRX :> rxTime;
        rxI = 0;
        rxTime += rxPeriod;
        break;
      case isRX => tmrRX when timerafter(rxTime) :> void :
        if (rxI < 8)
          RX :> >> rxByte;
        else { // receive stop bit
          RX :> void;
          putByte(rxByte >> 24);
          isRX = 0;
        }
        rxI++;
        rxTime += rxPeriod;
        break;
      case isTX => tmrTX when timerafter(txTime) :> void :
        if (txI < 8)
          TX <: >> txByte;
        else if (txI == 8)
          TX <: 1; // stop bit
        else
          isTX = 0;
        txI++;
        txTime += txPeriod;
        break;
} }  }
```

The variables `isTX`, `txI`, `isRX` and `rxI` determine which parts of the UART are active and how many bits of data have been transmitted and received.

The `while` loop first checks whether the transmitter is inactive with data available to transmit, in which case it outputs a start bit and sets the timeout for outputting the first data bit.

In the `select` statement, the guard

```
case !isRX => RX when pinseq(0) :> void :
```

checks whether `isRX` equals zero, indicating that the receiver is inactive, and if so it enables an input on the port `RX` when the value on its pins equals 0. The expression on the left of the operator => is said to *enable* the input. The body of this case sets a timeout for inputting the first data bit.

The second guard

```
case isRX => tmrRX when timerafter(rxTime) :> void :
```

checks whether `isRX` is non-zero, indicating that the receiver is active, and if so enables an input on the timer `tmrRX`. The body of this case inputs the next bit of data and, once all bits are input, it stores the data and sets `isRX` back to zero.

The third guard

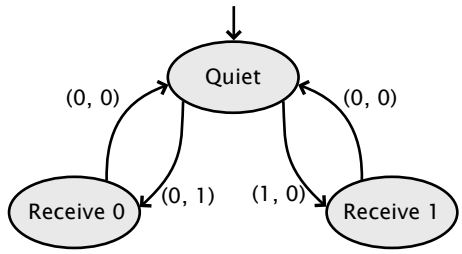
```
case isTX => tmrTX when timerafter(txTime) :> void :
```

checks whether `isTX` is non-zero, indicating that the transmitter is active, and if so enables an input on the timer `tmrTX`. The body of this case outputs the next bit of data and, once all bits are output, it sets `isTX` to zero.

If this UART controller is to be used in noisy environments, its reliability may be improved by sampling each input bit multiple times and averaging the result. A more robust implementation would also check that the stop bit received has an expected value of 1.

2.8 Parameterised Selection

Select statements can be implemented as functions, allowing their reuse in different contexts. One such example use is to parameterise the code used to sample data on a two-bit encoded line. As shown below, a quiet state on the line is represented by the value (0, 0), the value 0 is signified by a transition to (0, 1) and the value 1 is signified by a transition to (1, 0). Either of these transitions is followed by another transition back to (0, 0).



The program below makes use of a *select function* to input a single byte of data from two pins using this scheme.

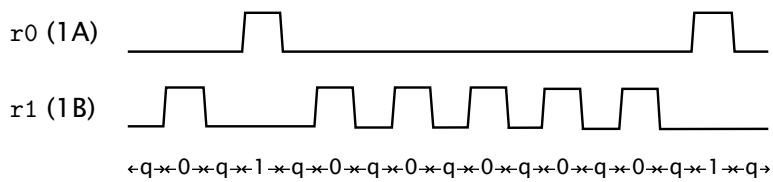
```
#include <xs1.h>

in port r0 = XS1_PORT_1A;
in port r1 = XS1_PORT_1B;

select inBit(in port r0, in port r1,
             int &x0, int &x1, char &byte) {
  case r0 when pinsneq(x0) :> x0 :
    if (x0 == 1)  /* transition to (1, 0) */
      byte = (byte << 1) | 1;
    break;
  case r1 when pinsneq(x1) :> x1 :
    if (x1 == 1)  /* transition to (0, 1) */
      byte = (byte << 1) | 0;
    break;
}

int main(void) {
  int x0 = 0, x1 = 0;
  char byte;
  for (int i=0; i<8; i++)
    inBit(r0, r1, x0, x1, byte);
}
```

The declaration

```
select inBit(in port r0, in port r1, int &x0, int &x1, char &byte)
```

declares `inBit` to be a select function that takes five arguments and has an implicit return type of `void`; its body contains two case statements.

The waveform diagram below shows example input stimuli for this program. The bit values received are 0, 1, 0, 0, 0, 0, 0 and 1 ('A').



In contrast to a UART, which transmits data at a fixed rate, this scheme allows for the fastest possible transmission supported by an XMOS device and the component to which it is connected.

A benefit of defining `inBit` as a select function is that its individual cases can be used to form part of a larger select statement, as in the program below which decodes a two-byte value sampled on four pins.

```
#include <xs1.h>
#define NBYTES 2

in port r[NBYTES*2] = { XS1_PORT_1A, XS1_PORT_1B,
                        XS1_PORT_1C, XS1_PORT_1D };
int main(void) {
  int state[NBYTES*2] = {0, 0, 0, 0};
  char byte[NBYTES];
  for (int i=0; i<8*NBYTES; i++)
    select {
      case inBit(r[0], r[1], state[0], state[1], byte[0]);
      case inBit(r[2], r[3], state[2], state[3], byte[1]);
    }
}
```

The `select` statement calls the function `inBit` in two of its `case` statements, causing the processor to enable events on the ports passed to it as arguments.

A more concise way to specify the top-level select statement is to use a *replicator*, as in:

```
select {
  case (int i=0; i<NBYTES; i++)
    inBit(r[i*2], r[i*2+1], state[i*2], state[i*2+1], byte[i]);
}
```

The replicator

```
(int i=0; i<2; i++)
```

iterates twice, each time calling the select function `inBit`, which enables the ports indexed by different values of `i`. The number of iterations need not be constant, but the iterator must not be modified outside of the replicator.

CHAPTER 3

Concurrency

Many designs require of a collection of tasks to be performed at the same time. Some of these tasks may perform independent activities, while others engage with one another to complete shared objectives. XC provides simple mechanisms for creating *concurrent threads* that can run independently and interact with one another on demand. Data is communicated between threads using *channels*, which provide point-to-point connections between pairs of threads. Channels can be used to communicate data either synchronously or asynchronously.

3.1 Creating Concurrent Threads

The program below creates four concurrent threads, all of which run separate tasks independently of one another. Two of these threads are executed on XCore 0, one on XCore 1 and one on XCore 2.

```
#include <platform.h>

on stdcore[0] : out port tx      = XS1_PORT_1A;
on stdcore[0] : in  port rx      = XS1_PORT_1B;
on stdcore[1] : out port lcdData = XS1_PORT_32A;
on stdcore[2] : in  port keys    = XS1_PORT_8B;

int main(void) {
  par {
    on stdcore[0] : uartTX(tx);
    on stdcore[0] : uartRX(rx);
    on stdcore[1] : lcdDrive(lcdData);
    on stdcore[2] : kbListen(keys);
  }
}
```

The header file `platform.h` provides a declaration of the global variable `stdcore`, which is used to specify the locations of ports and threads.[1]

The declaration

```
on stdcore[0] : out port p = XS1_PORT_1A;
```

declares a 1-bit output port named `p` that refers to the port identifier 1A on standard core number 0.

The four statements inside the braces of the `par` are run concurrently as four separate threads using *fork-join parallelism*: at the opening brace `{` the parent creates three more threads; each of these threads then executes a function; and at the closing brace `}` the parent waits for all functions to return before continuing.

`par` statements may be used anywhere in a program. Each XS1 device has a limit of eight threads available on each of its processors, and a program that attempts to exceed this limit is invalid.

The `on` statement is used to specify the physical location of components connected to ports and to partition a collection of threads between the available XCores.

For single-core programs, none of the port declarations need be prefixed with `on`, in which case all ports and threads are placed on XCore 0. For multicore programs, all ports and threads must be explicitly prefixed with `on`.

A multicore `main` function may contain only channel declarations, a single `par` statement and an optional `return` statement. The `on` statement may be used to specify the location of threads only within this function.

3.2 Thread Disjointness Rules

All variables are subject to usage rules that prevent them from being shared by threads in potentially dangerous ways. In general, each thread has full access to its own private variables, but limited access to variables that are shared with other threads. The rules for disjointness on a set of threads $T_0 \ldots T_i$ and a set of variables $V_0 \ldots V_j$ are as follows:

- If thread T_x contains any modification to variable V_y then none of the other threads $(T_t, t \neq x)$ are allowed to use V_y.
- If thread T_x contains a reference to variable V_y then none of the other threads $(T_t, t \neq x)$ are allowed to modify V_y.
- If thread T_x contains a reference to port V_p then none of the other threads are allowed to use V_p.

[1] The target platform is described using the XMOS network specfication language XN. Most board support packages provide a corresponding XN file, which describes the available devices and their connectivity. This data is used during the mapping stage of compilation to product a multi-node executable file that can boot and configure the entire system.

In other words, a group of threads can have *shared* read-only access to a variable, but only a single thread can have *exclusive* read-write access to a variable. These rules guarantee that each thread has a well-defined meaning that is independent of the order in which instructions in other threads are scheduled. Interaction between threads takes place explicitly using inputs and outputs on channels (see §3.3).

3.2.1 Examples

The example program below is legal, since `k` is shared read-only in threads X and Y, `i` is modified in X and not used in Y, and `j` is modified in Y and not used in X.

```
int main(void) {
  int i = 1, j = 2, k = 3;
  par {
    i = k + 1;  // Thread X
    j = k - 1;  // Thread Y
  }
}
```

If either `i` or `j` is also read in another thread, the example becomes illegal, as shown in the program below.

```
int main(void) {
  int i = 1, j = 2, k;
  par {
    i = j + 1;  // Thread X: illegal sharing of i
    k = i - 1;  // Thread Y: illegal sharing of i
  }
}
```

This program is ambiguous since the value of `i` read in thread Y depends upon whether the assignment to `i` in thread X has already happened or not.

The program below is legal, since `a[0]` is modified in thread X and not used in thread Y, and `a[1]` is modified in Y and not used in X.

```
int main(void) {
  int a[2];
  par {
    a[0] = f(0); // Thread X
    a[1] = f(1); // Thread Y
  }
}
```

The program below is illegal since `a[1]` is modified in thread *X* and an unknown element of `a` is modified in thread *Y*.

```
int x;
int main(void) {
  int a[10];
  par {
    a[1] = f(1); // Thread X: illegal sharing of x[1]
    a[x] = f(x); // Thread Y: illegal sharing of x[1]
  }
}
```

In general, indexing an array by anything other than a constant value is treated as if all elements in the array are accessed.

The program below is illegal since the array `a` is passed by reference to the function `f` in both threads *X* and *Y*, which may possibly modify its value.

```
void f(int[]);

int main(void) {
  int a[10];
  par {
    f(a); // Thread X: illegal sharing of a
    f(a); // Thread Y: illegal sharing of a
  }
}
```

If `f` does not modify the array then its parameter should be declared with `const`, which would make the above program legal.

The disjointness rules apply individually to parallel statements in sequence, and recursively to nested parallel statements. The example program below is legal.

```
int main(void) {
  int i = 1, j = 2, k = 3;
  par {
    i = k + 1;  // Thread X
    j = k - 1;  // Thread Y
  }
  i = i + 1;
  par {
    j = i - 1;  // Thread U
    k = i + 1;  // Thread V
  }
}
```

In this example, `i` is first declared and initialised in the main thread; it is then used exclusively in thread *X* (thread *Y* is not allowed access). Once *X* and *Y* have joined, `i` it is used again by the main thread; finally it is shared bewteen threads *U* and *V*.

3.3 Channel Communication

A *channel* provides a synchronous, point-to-point connection between two threads over which data may be communicated. The program below uses a channel to communicate data from a producer thread on one processor to a consumer thread on another.

```
#include <platform.h>

on stdcore[0] : out port tx   = XS1_PORT_1A;
on stdcore[1] : in  port keys = XS1_PORT_8B;


void uartTX(chanend dataIn, port tx) {
  char data;
  while (1) {
    dataIn :> data;
    transmitByte(tx, data);
  }
}

void kbListen(chanend c, port keys) {
  char data;
  while (1) {
    data = waitForKeyStroke(keys);
    c <: data;
  }
}

int main(void) {
  chan c;
  par {
    on stdcore[0] : uartTX(c, tx);      // Thread X
    on stdcore[1] : kbListen(c, keys); // Thread Y
  }
}
```

The declaration

```
void uartTX(chanend dataIn, port tx)
```

declares `uartTX` to be a function that takes a channel end and a port as its arguments.

The declaration

```
void kbListen(chanend c, port keys);
```

declares `kbListen` to be a function that takes a channel end and a port as its argument.

In the function `main`, the declaration

```
chan c;
```

declares a channel. The channel is used in two threads of a `par` and each use implicitly refers to one of its two channel ends. This usage establishes a link between thread X on XCore 0 and thread Y on XCore 1.

Thread X calls the function `uartTX`, which receives data over a channel and outputs it to a port. Thread Y calls `kbListen`, which waits for keyboard strokes from a port and outputs the data on a channel to the UART transmitter on thread X. As the channel is synchronous, when `kbListen` outputs data, it waits until `uartTX` is ready to receive the data before continuing.

Channels are lossless, which means that data output in one thread is guaranteed to be delivered for input by another thread. Each output in one thread must therefore be matched by an input in another, and the amount of data output must equal the amount input or else the program is invalid.

3.3.1 Channel Disjointness Rules

The rules for disjointness on a set of threads $T_0 \ldots T_i$ and a set of channels $C_0 \ldots C_j$ are as follows:

- If threads T_x and T_y (where $x \neq y$) contain a use of channel C_y then none of the other threads $(T_t, t \neq x, y)$ are allowed use C_y.
- If thread T_x contains a use of channel end C_y then none of the other threads $(T_t, t \neq x)$ are allowed to use C_y.

In other words, each channel can be used in at most two threads. If a channel is used in only one thread then attempting to input or output on the channel will block forever.

The disjointness rules for variables and channels together guarantee that any two threads can be run concurrently on any two processors, subject to a physical route existing between the processors. As a general rule, threads that interact with one another frequently should usually be located close together.

3.4 Transactions

Input and output statements on channels usually synchronise the communication of data. This is not always desirable, however, as it disrupts the flow of the program, causing threads to block. The time taken to synchronise, including the time spent idle while blocking, can reduce overall performance.

In XC it is possible for two threads to engage in a *transaction*, in which a sequence of matching outputs and inputs are communicated over a channel asynchronously, with the entire transaction being synchronised at its beginning and end. As with

individual channel communications, the total amount of data output must equal the total amount input.

The program below uses a transaction to communicate a packet of data between two threads efficiently.

```
#include <platform.h>

int snd[3], rcv[3];

int main(void) {
  chan c;
  par {
    on stdcore[0] : master {   // Thread X
      for (int i=0; i<10; i++)
        c <: snd[i];
    }
    on stdcore[1] : slave {    // Thread Y
      for (int i=0; i<10; i++)
        c :> rcv[i];
} } }
```

A transaction consists of a *master* thread and a *slave* thread running concurrently. The threads first synchronise upon entry to the master and slave blocks. Ten integer values are then communicated asynchronously: thread X blocks only if data can no longer be dispatched (due to the channel buffering being full), and thread Y blocks only if there is no data available. Finally, the threads synchronise upon exiting the master and slave blocks.

Each transaction is permitted to communicate on precisely one channel. This ensures that deadlocks do not arise due to an output on one channel blocking as a result of a switch being full with incoming data that is not yet ready to be received.

The program below defines the body of a transaction as a function, which is called as the master component of a communication.

```
transaction inArray(chanend c, int data[], int size) {
  for (int i=0; i<size; i++)
    c :> data[i];
}

int main(void) {
  chan c;
  int snd[3], rcv[3];
  par {
    master inArray(c, rcv, 3);
    slave {
      for (int i=0; i<10; i++)
        c :> rcv[i];
} } }
```

The declaration

```
transaction inArray(chanend c, char data[], int size)
```

declares `inArray` to be a transaction function that takes one end of a channel, an array of integers and the size of the array. A transaction function must declare precisely one channel end parameter.

In `main`, the call to `inArray` is prefixed with `master`, which specifies that the function is called as a master that communicates with a slave.

A slave statement may be used in the guard of a `select` statement, as in:

```
select {
  case slave { inArray(c1, packet, P_SIZE); } :
    process(packet);
    break;
  case slave { inArray(c2, packet, P_SIZE); } :
    process(packet);
    break;
}
```

A master operation by definition commits to completing and is therefore disallowed from appearing in a guard.

3.5 Streams

A *streaming channel* establishes a permanent route between two threads over which data can be efficiently communicated without synchronisation. The program below consists of three threads that together input a stream of data from a port, filter the data and output it to another port.

```
#include <platform.h>

on stdcore[0] : port lineIn = XS1_PORT_8A;
on stdcore[1] : port spkOut = XS1_PORT_8A;

int main(void) {
  streaming chan s1, s2;
  par {
    on stdcore[0] : audioRcv(lineIn, s1);
    on stdcore[0] : BiQuadFilter(s1,s2);
    on stdcore[1] : audioSnd(spkOut, s2);
} }
```

The declaration

```
streaming chan s1, s2;
```

declares `s1` and `s2` as channels that transport data without performing any synchronisation. A route is established for the stream at its declaration and is closed down when the declaration goes out of scope.

Streaming channels provide the fastest possible data rates. An output statement takes just a single instruction to complete and is dispatched immediately as long as there is space in the channel's buffer. An input statement takes a single instruction to complete and blocks only if the channel buffer's is empty. In contrast to transactions, multiple streams can be processed concurrently, but there is a limit to how many streaming channels can be declared together as streams established between XCores require capacity to be reserved in switches. This limit does not apply to channels and transactions.

3.6 Parallel Replication

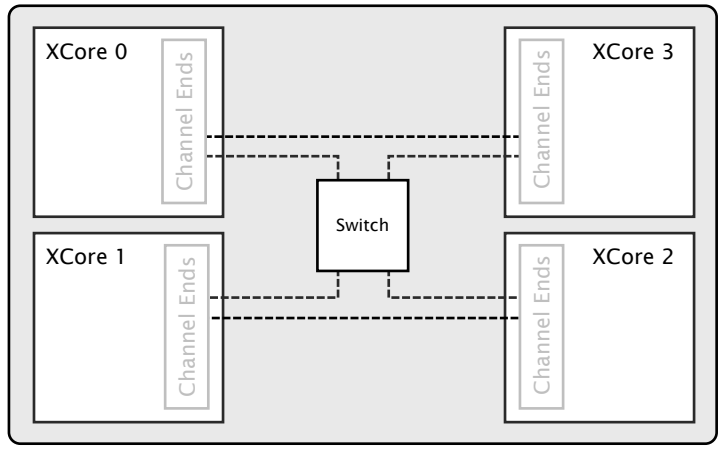
A *replicator* provides a concise and simple way to implement concurrent programs in which a collection of nodes perform the same operation on different datasets. The program below constructs a communications network between four nodes running on four different threads.

```
#include <platform.h>

port p[4] = {
  on stdcore[0] : XS1_PORT_1A,
  on stdcore[1] : XS1_PORT_1A,
  on stdcore[2] : XS1_PORT_1A,
  on stdcore[3] : XS1_PORT_1A
};

void node(chanend, chanend, port, int n);

int main(void) {
  chan c[4];
  par (int i=0; i<4; i++)
    on stdcore[i] : node(c[i], c[(i+1)%4], p[i], i);
  return 0;
}
```

The replicator

```
(int i=0; i<4; i++)
```

executes four bodies of code, each containing an instance of the function `node` on a different thread. The number of iterations must be constant, and the iterator must not be modified outside of the replicator. The communication network established by this program is illustrated on the next page.



The structure of this program is similar to a token ring network, in which each thread inputs a token from one of its neighbours, performs an action and then outputs the token to its other neighbour.

3.7 Services

An XMOS network can interface with any device that implements the XMOS Link protocol. The program below communicates with an FPGA service connected to the network.

```
#include <platform.h>
port p = XS1_PORT_1A;

void inData(chanend c, port p) {
  // ... input data from p and output to c
}

service fpgaIF(chanend);

int main(void) {
  chan c;
  par {
    inData(c, p);
    fpgaIF(c);
} }
```

The declaration

```
service fpgaIF(chanend);
```

declares `fpgaIF` to be a service available on the XMOS network. A function declared as a service may contain only channel-end parameters and must not be given a

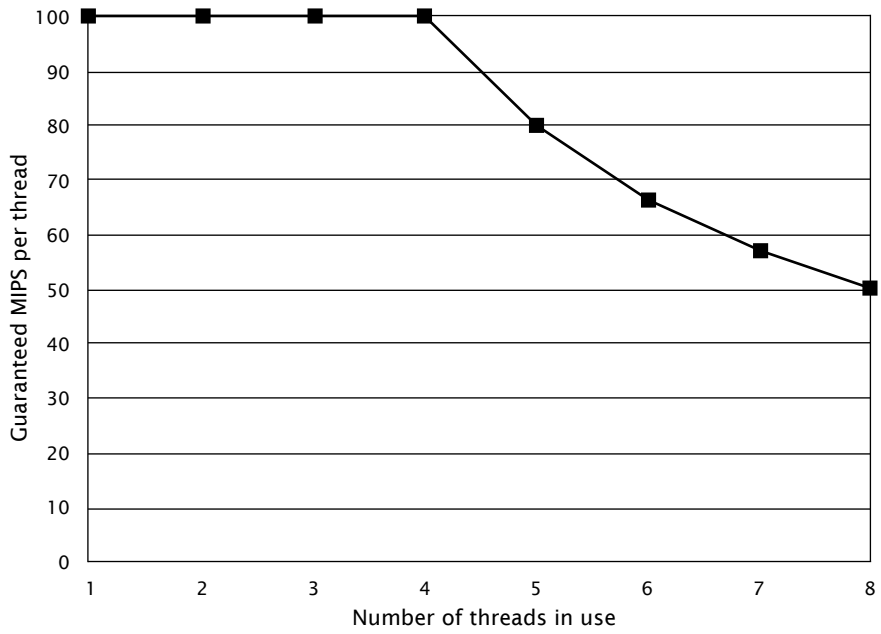
definition. The characteristics of the links used to implement the channel ends are defined in the XN file.

Another use of services is for interfacing with functions pre-programmed into the non-volatile memory of an XCore by a third-party manufacturer. Typically, the manufacturer provides an XN file that contains all service declarations, which are available in the file `<platform.h>`.

3.8 Thread Performance

The XMOS architecture is designed to perform multiple real-time tasks concurrently, each of which is guaranteed predictable thread performance. Each processor uses a round-robin thread scheduler, which guarantees that if up to four threads are active, each thread is allocated a quarter of the processing cycles. If more than four threads are active, each thread is allocated at least $\frac{1}{n}$ cycles (for n threads). The minimum performance of a thread can therefore be calculated by counting the number of concurrent threads at a specific point in the program.

The graph below shows the guaranteed performance obtainable from each thread on a 400MHz XCore, depending on the total number of threads in use.



Because individual threads may be delayed on I/O, their unused processor cycles can be taken by other threads. Thus, for more than four threads, the performance of each thread is often higher than the minimum shown above.

CHAPTER 4

Clocked Input and Output

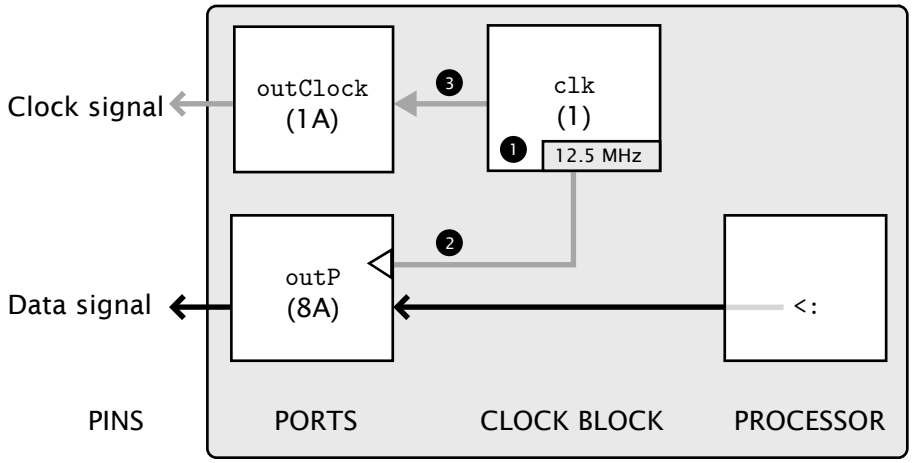
Many protocols require data to be sampled and driven on specific edges of a clock. Ports can be configured to use either an internally generated clock or an externally sourced clock, and the processor can record and control on which edges each input and output operation occurs. In XC, these operations can be directly expressed in the input and output statements using the *timestamped* and *timed* operators.

4.1 Generating a Clock Signal

The program below configures a port to be clocked at a rate of 12.5MHz, outputting the corresponding clock signal with its output data.

```
#include <xs1.h>

out port outP     = XS1_PORT_8A;
out port outClock = XS1_PORT_1A;
clock    clk      = XS1_CLKBLK_1;

int main(void) {
  configure_clock_rate(clk, 100, 8);
  configure_out_port(outP, clk, 0);
  configure_port_clock_output(outClock, clk);
  start_clock(clk);

  for (int i=0; i<5; i++)
    outP <: i;
}
```

The program configures the ports `outP` and `outClock` as illustrated below.



The declaration

```
clock clk = XS1_CLKBLK_1;
```

declares a clock named `clk`, which refers to the *clock block* identifier `XS1_CLKBLK_1`. Clocks are declared as global variables, with each declaration initialised with a unique resource identifier.

❶ The statement

```
configure_clock_rate(clk, 100, 8);
```

configures the clock `clk` to have a rate of 12.5MHz. The rate is specified as a fraction (100/8) because XC only supports integer arithmetic types.

❷ The statement

```
configure_out_port(outP, clk, 0);
```

configures the output port `outP` to be clocked by the clock `clk`, with an initial value of 0 driven on its pins.

❸ The statement

```
configure_port_clock_output(outClock, clk)
```

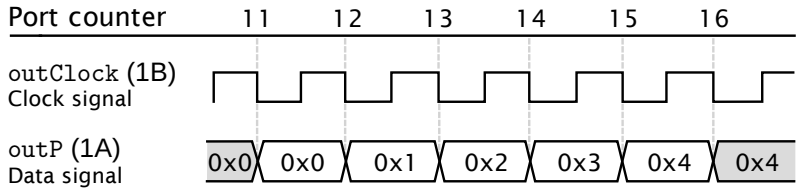
causes the clock signal `clk` to be driven on the pin connected to the port `outClock`, which a receiver can use to sample the data driven by the port `outP`.

The statement

```
start_clock(clk);
```

causes the clock block to start producing edges.

A port has an internal 16-bit counter, which is incremented on each falling edge of its clock. The waveform diagram below shows the port counter, clock signal and data driven by the port.



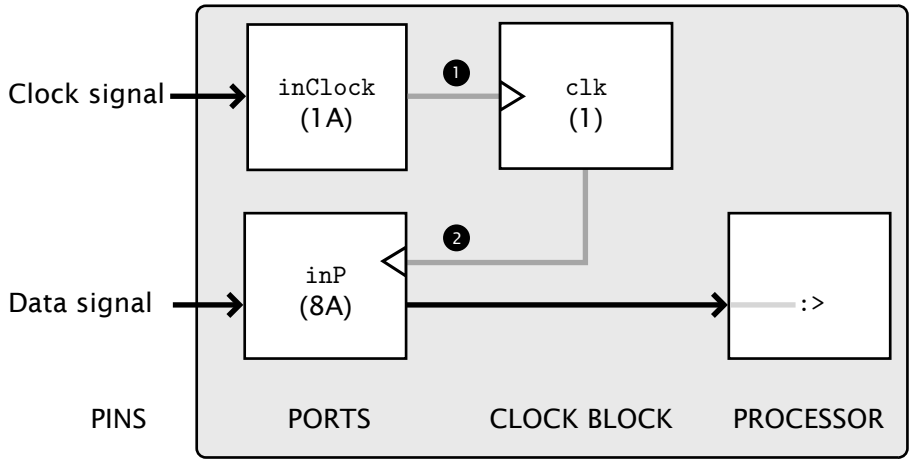
An output by the processor causes the port to drive output data on the next falling edge of its clock; the data is held by the port until another output is performed.

4.2 Using an External Clock

The following program configures a port to synchronise the sampling of data to an external clock.

```
#include <xs1.h>

in port inP     = XS1_PORT_8A;
in port inClock = XS1_PORT_1A;
clock   clk     = XS1_CLKBLK_1;

int main(void) {
  configure_clock_src(clk, inClock);
  configure_in_port(inP, clk);

  start_clock(clk);
  for (int i=0; i<5; i++)
    inP :> int x;
}
```

The program configures the ports `inP` and `inClock` as illustrated below.



❶ The statement

```
configure_clock_src(clk, inClock);
```

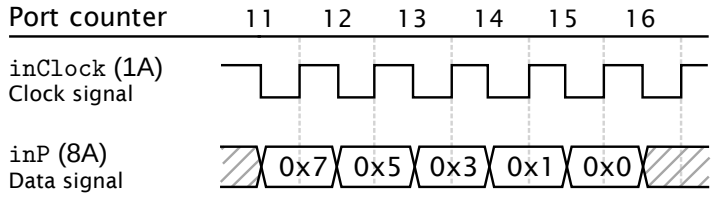
configures the 1-bit input port `inClock` to provide edges for the clock `clk`. An edge occurs every time the value sampled by the port changes.

❷ The statement

```
configure_in_port(inP, clk);
```

configures the input port `inP` to be clocked by the clock `clk`.

The waveform diagram below shows the port counter, clock signal, and example input stimuli.



An input by the processor causes the port to sample data on the next rising edge of its clock. The values input are 0x7, 0x5, 0x3, 0x1 and 0x0.

4.3 Performing I/O on Specific Clock Edges

It is often necessary to perform an I/O operation on a port at a specific time with respect to its clock. The program on the facing page drives a pin high on the third clock period and low on the fifth.

```
void doToggle(out port toggle) {
  int count;
  toggle <: 0 @ count;    // timestamped output
  while (1) {
    count += 3;
    toggle @ count <: 1;  // timed output
    count += 2;
    toggle @ count <: 0;  // timed output
  }
}
```

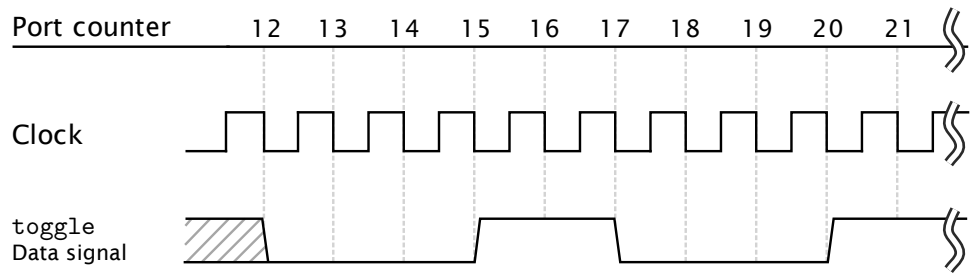
The statement

```
toggle <: 0 @ count;
```

performs a *timestamped output*, outputting the value 0 to the port `toggle` and reading into the variable `count` the value of the port counter when the output data is driven on the pins. The program then increments `count` by a value of 3 and performs a *timed output* statement

```
toggle @ count <: 1;
```

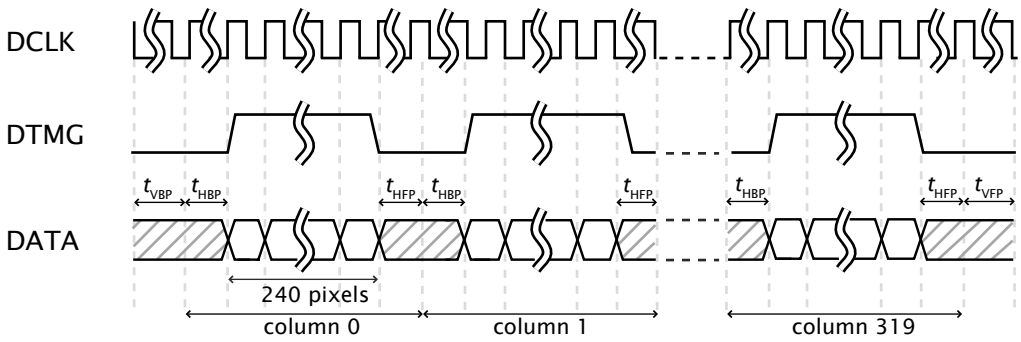
This statement causes the port to wait until its counter equals the value `count+3` (advancing three clock periods) and to then drive its pin high. The last two statements delay the next output by two clock periods. The waveform diagram below shows the port counter, clock signal and data driven by the port.

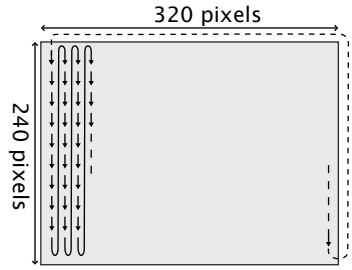


The port counter is incremented on the falling edge of the clock. On intermediate edges for which no value is provided, the port continues to drive its pins with the data previously output.

4.4 Case Study: LCD Screen Driver

LCD screens are found in many embedded systems. The principal method of driving most screens is the same, although the specific details vary from screen to screen. The diagram on the following page illustrates the operation of a Hitachi TX14 series screen, including the waveform requirements for transmitting a single frame of video [4].





Clock cycles	
t_{VBP}	2310
t_{VFP}	1925
t_{HBP}	30
t_{HFP}	25

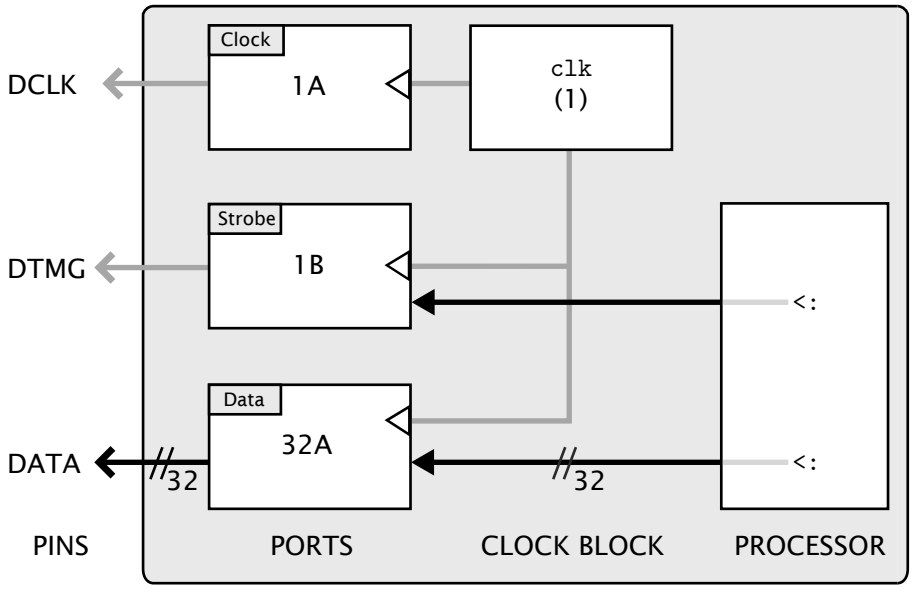
Table 1

The screen has a resolution of 320x240 pixels. It requires pixel data to be provided in column order with each value driven on a specific edge of a clock. The signals are as follows:

- DCLK is a clock signal generated by the driver, which must be configured within the range of 4.85MHz to 7.00MHz. The value chosen determines the screen refresh rate.
- DTMG is a data valid signal which must be driven high whenever data is transmitted.
- DATA carries 18-bit RGB pixel data to the screen.

The specification requires that pixel values for each column are driven on consecutive cycles with a 55 cycle delay between each column and a 4235 cycle delay between each frame (see Table 1).

LCD screens are usually driven by dedicated hardware components due to their clocking requirements. Implementing an LCD screen driver in XC is easy due to the clock synchronisation supported by the XMOS architecture. The required port configuration is illustrated on the next page.



The ports `DATA` and `DTMG` are both clocked by an internally generated clock, which is made visible on the port `DCLK`. The program below defines a function that configures the ports in this way.

```
#include <xs1.h>

out port DCLK = XS1_PORT_1A;
out port DTMG = XS1_PORT_1B;
out port DATA = XS1_PORT_32A;
clock    clk  = XS1_CLKBLK_1;

void lcdInit(void) {
  configure_clock_rate(clk, 100, 17); // 100/17 = 5.9Mhz
  configure_out_port(DATA,  clk, 0);
  configure_out_port(DTMG, clk, 0);
  configure_port_clock_output(DCLK, clk);
  start_clock(clk);
}
```

The clock rate specified is 5.9Mhz. The time required to transmit a frame is $320 * 240 + 240 * 55 + 4235 = 94235$ clock ticks, giving a frame rate of $\frac{5.9}{94235}$ = 62Hz. The function on the following page outputs a sequence of pixel values to the LCD screen on the clock edges required by the specification.

```
void lcdDrive(streaming chanend c, out port DATA,
              out port DTMG) {
  unsigned x, time;
  DTMG <:0 @ time;
  while (1) {
    time += 4235;
    for (int cols=0; cols<320; cols++) {
      time +=30;
      c :> x;
      DTMG @ time <: 1;      // strobe high
      DATA @ time <: x;      // pixel 0
      for (int rows=1; rows<240; rows++) {
        c :> x;
        DATA <: x;           // pixels 1..239
      }
      DTMG @ time+240 <: 0; // strobe low
      time += 25;
} } }
```

A stream of data is input from a channel end. The body of the `while` loop transmits a single frame and the body of the outer `for` transmits each column. The program instructs the port `DTMG` to start driving its pin high when it starts outputting a column of data and to stop driving afterwards.

An alternate solution is to configure the port `DATA` to generate a ready-out strobe signal on `DTMG` (see §6.4) and to remove the two outputs to `DTMG` by the processor in the source code.

4.5 Summary of Clocking Behaviour

The semantics for inputs and outputs on clocked (unbuffered) ports are summarised as follows.

Output Statements

- An *output* causes data to be driven on the next falling edge of the clock. The output blocks until the subsequent rising edge.
- A *timed output* causes data to be driven by the port when its counter equals the specified time. The output blocks until the next rising edge after this time.
- The data driven on one edge continues to be driven on subsequent edges for which no new output data is provided.

Input Statements

- An *input* causes data to be sampled by the port on the next rising edge of its clock. The input blocks until this time.

- A *timed input* causes data to be sampled by the port when its counter equals the specified time. The input blocks until this time.
- A *conditional input* causes data to be sampled by the port on each rising edge until the sampled data satisfies the condition. The input blocks until this time, taking the most recent data sampled.

Select Statements

A select statement waits for any one of the ports in its cases to become ready and completes the corresponding input operation, where:

- For an *input*, the port is ready at most once per period of its clock.
- For a *timed input*, the port is ready only when its counter equals the specified time.
- For a *conditional input*, the port is ready only when the data sampled satisfies the condition.
- For a *timed conditional input*, the port is ready only when its counter is equal or greater than the specified time and the value sampled satisfies the condition.

For a timestamped operation that records the value t, the next possible time that the thread can input or output is $t + 1$.

On XS1 devices, all ports are buffered (see §C.1). The resulting semantics, which extend those given above, are discussed in the next chapter.

CHAPTER 5

Port Buffering

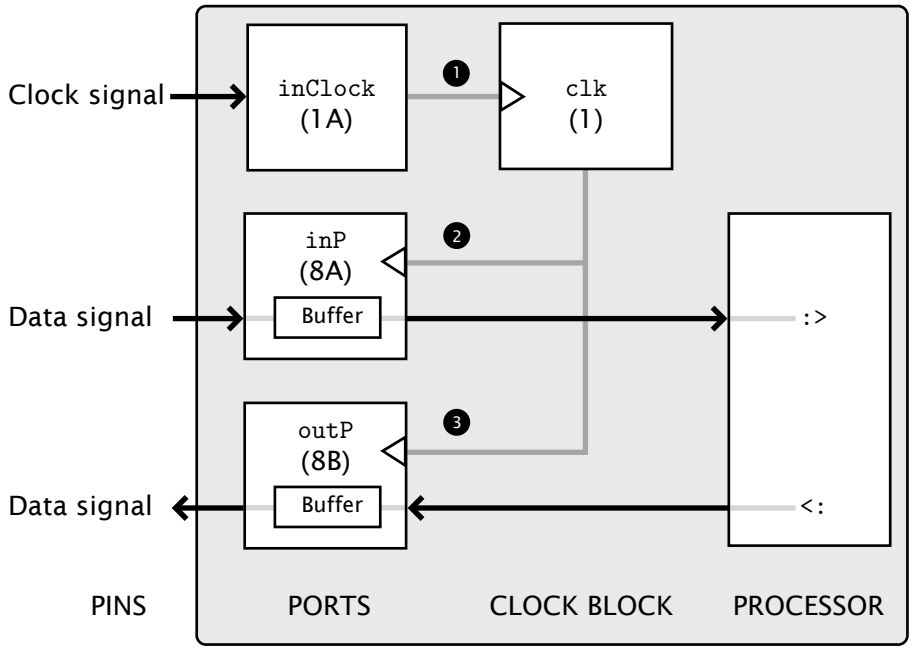
The XMOS architecture provides buffers that can improve the performance of programs that perform I/O on clocked ports. A buffer can hold data output by the processor until the next falling edge of the port's clock, allowing the processor to execute other instructions during this time. It can also store data sampled by a port until the processor is ready to input it. Using buffers, a single thread can perform I/O on multiple ports in parallel.

5.1 Using a Buffered Port

The following program uses a buffered port to decouple the sampling and driving of data on ports from a computation.

```
#include <xs1.h>

in  buffered port:8 inP     = XS1_PORT_8A;
out buffered port:8 outP    = XS1_PORT_8B;
in port             inClock = XS1_PORT_1A;
clock               clk     = XS1_CLKBLK_1;

int main(void) {
  configure_clock_src(clk, inClock);
  configure_in_port(inP, clk);
  configure_out_port(outP, clk, 0);
  start_clock(clk);
  while (1) {
    int x;
    inP  :> x;
    outP <: x + 1;
    f();
} }
```

The program configures the ports `inP`, `outP` and `inClock` as illustrated below.



The declaration

```
in buffered port:8 inP = XS1_PORT_8A;
```

declares a buffered input port named `inP`, which refers to the 8-bit port identifier 8A.

❶ The statement

```
configure_clock_src(clk, inClock);
```

configures the 1-bit input port `inClock` to provide edges for the clock `clk`.

❷ The statement

```
configure_in_port(inP, clk);
```

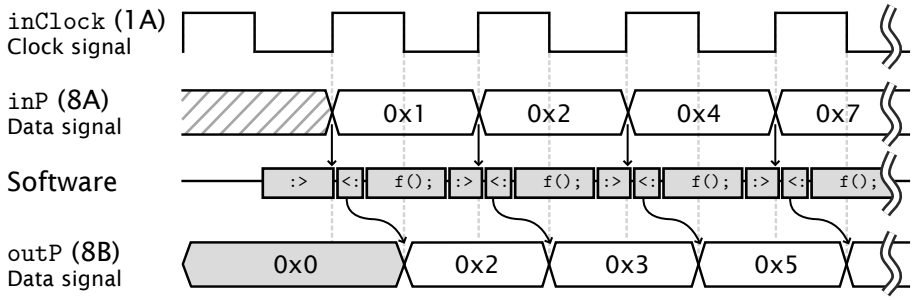
configures the input port `inP` to be clocked by the clock `clk`.

❸ The statement

```
configure_out_port(outP, clk, 0);
```

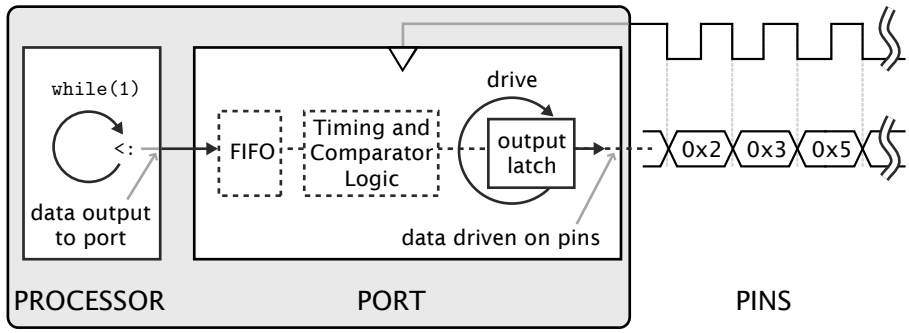
configures the output port `outP` to be clocked by the clock `clk`, with an initial value of 0 driven on its pins.

The waveform diagram below shows example input stimuli and expected output for this program. It also shows the relative waveform of the statements executed in the `while` loop by the processor.



The first three values input are 0x1, 0x2 and 0x4, and in response the values output are 0x2, 0x3 and 0x5.

The diagram below illustrates the buffering operation in the hardware.



The diagram shows the processor executing the `while` loop that outputs data to the port. The port buffers this data so that the processor can continue executing subsequent instructions while the port drives the data previously output for a complete period. On each falling edge of the clock, the port takes the next byte of data from its buffer and drives it on its pins. As long as the instructions in the loop execute in less time than the port's clock period, a new value is driven on the pins on every clock period.

The fact that the first input statement is executed before a rising edge means that the input buffer is not used. The processor is always ready to input the next data before it is sampled, which causes the processor to block, effectively slowing itself down to the rate of the port. If the first input occurs after the first value is sampled, however, the input buffer holds the data until the processor is ready to accept it and each output blocks until the previously output value is driven.

Timed operations represent time in the future. The waveform and comparitor logic allows timed outputs to be buffered, but for timed and conditional inputs the buffer is emptied before the input is performed.

5.2 Synchronising Clocked I/O on Multiple Ports

By configuring more than one buffered port to be clocked from the same source, a single thread can cause data to be sampled and driven in parallel on these ports. The program below first synchronises itself to the start of a clock period, ensuring the maximum amount of time before the next falling edge, and then outputs a sequence of 8-bit character values to two 4-bit ports that are driven in parallel.

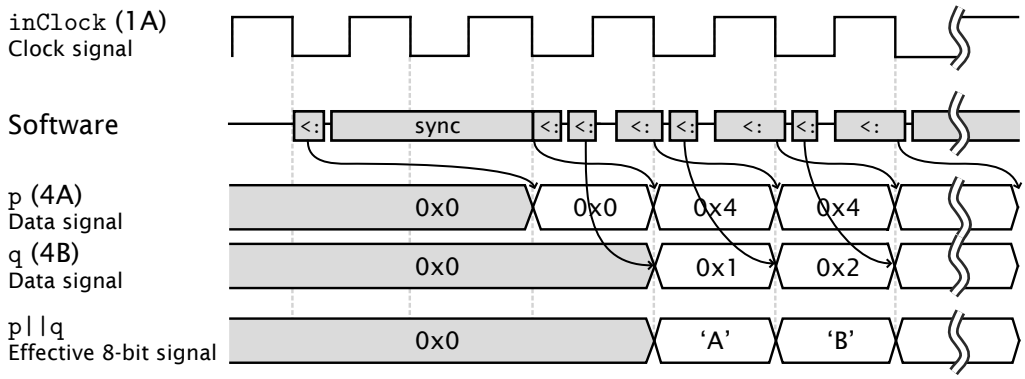
```
#include <xs1.h>

out buffered port p:4     = XS1_PORT_4A;
out buffered port q:4     = XS1_PORT_4B;
in             port inClock = XS1_PORT_1A;
clock          clk          = XS1_CLKBLK_1;

int main(void) {

  configure_clock_src(clk, inClock);
  configure_out_port(p, clk, 0);
  configure_out_port(q, clk, 0);
  start_clock(clk);

  p <: 0;  // start an output
  sync(p); // synchronise to falling edge

  for (char c='A'; c<='Z'; c++) {
    p <: (c & 0xF0) >> 4;
    q <: (c & 0x0F);
  }
}
```

The statement

```
sync(p);
```

causes the processor to wait until the next falling edge on which the last data in the buffer has been driven for a full period, ensuring that the next instruction is executed just after a falling edge. This ensures that the subsequent two output statements in the loop are both executed in the same clock period.

The diagram on the next page shows the data output by the processor and driven by the two ports.



The recommended way to synchronise to a rising edge is to clear the buffer using the standard library function `clearbuf` and then perform an input.

5.3 Summary of Buffered Behaviour

The semantics for I/O on clocked buffered ports are summarised as follows.

Output Statements

- An output inserts data into the port's FIFO. The processor waits only if the FIFO is full.
- At most one data value is removed from the FIFO and driven by the port per period of its clock.
- A timed output inserts data into the port's FIFO for driving when the port counter equals the specified time. The processor waits only if the FIFO is full.
- A timestamped output causes the processor to wait until the output is driven (required to determine the timestamp value).
- The data driven on one edge continues to be driven on subsequent edges.

Input Statements

- At most one value is sampled by the port and inserted into its FIFO per period of its clock. If the FIFO is full, its oldest value is dropped to make room for the most recently sampled value.
- An input removes data from a port's FIFO. The processor waits only if the FIFO is empty.
- Timed and conditional inputs cause any data in the FIFO to be discarded and then behave as in the unbuffered case.

CHAPTER 6

Serialisation and Strobing

The XMOS architecture provides hardware support for operations that frequently arise in communication protocols. A port can be configured to perform *serialisation*, useful if data must be communicated over ports that are only a few bits wide (such as in §2.5), and *strobing*, useful if data is accompanied by a separate data valid signal (such as in §4.4). Offloading these tasks to the ports frees up more processor time for executing computations.

6.1 Serialising Output Data using a Port

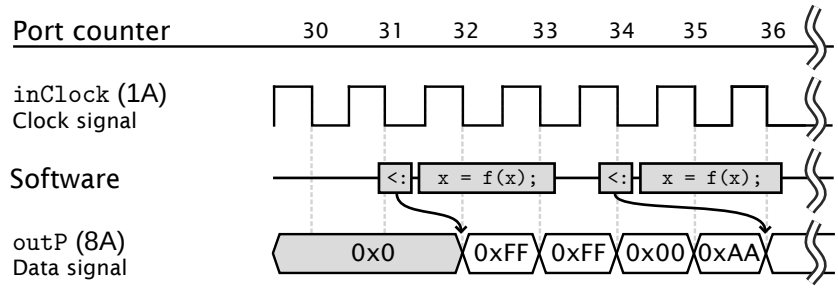
A clocked port can serialise data, reducing the number of instructions required to perform an output. The program below outputs a 32-bit value onto 8 pins, using a clock to determine for how long each 8-bit value is driven.

```
#include <xs1.h>

out buffered port:32 outP    = XS1_PORT_8A;
in           port    inClock = XS1_PORT_1A;
clock                clk     = XS1_CLKBLK_1;

int main(void) {
  int x = 0xFFFF00AA;
  configure_clock_src(clk, inClock);
  configure_out_port(outP, clk, 0);
  start_clock(clk);

  while (1) {
    outP <: x;
    x = f(x);
  }
}
```

The declaration

```
out buffered port:32 outP = XS1_PORT_8A;
```

declares the port `outP` to drive 8 pins from a 32-bit *shift register*. The type `port:32` specifies the number of bits that are transferred in each output operation (the *transfer width*). The initialisation `XS1_PORT_8A` specifies the number of physical pins connected to the port (the *port width*). The waveform diagram below shows the data driven by this program.



By offloading the serialisation to the port, the processor has only to output once every 4 clock periods. On each falling edge of the clock, the least significant 8 bits of the shift register are driven on the pins; the shift register is then right-shifted by 8 bits.

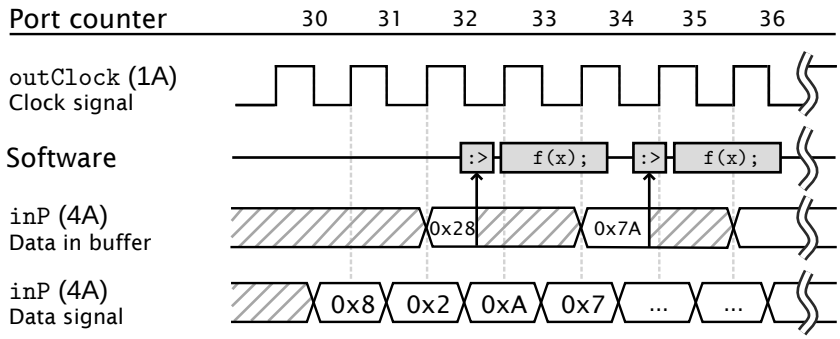
On XS1 devices, ports used for serialisation must be qualified with the keyword `buffered`; see §C.1 for further explanation.

6.2 Deserialising Input Data using a Port

A port can deserialise data, reducing the number of instructions required to input data. The program below performs a 4-to-8 bit conversion on an input port, controlled by a 25MHz clock.

```
#include <xs1.h>
in  buffered port:8 inP      = XS1_PORT_4A;
out          port   outClock = XS1_PORT_1A;
clock               clk25    = XS1_CLKBLK_1;

int main(void) {
  configure_clock_rate(clk25, 100, 4);
  configure_in_port(inP, clk25);
  configure_port_clock_output(outClock, clk25);
  start_clock(clk25);
  while (1) {
    int x;
    inP :> x;
    f(x);
} }
```

The program declares `inP` to be a 4-bit wide port with an 8-bit transfer width, meaning that two 4-bit values can be sampled by the port before they must be input by the processor. As with output, the deserialiser reduces the number of instructions required to obtain the data. The waveform diagram below shows example input stimuli and the period during which the data is available in the port's buffer for input.



Data is sampled on the rising edges of the clock and, when shifting, the least significant nibble is read first. The sampled data is available in the port's buffer for input for two clock periods. The first two values input are 0x28 and 0x7A.

6.3 Inputting Data Accompanied by a Data Valid Signal

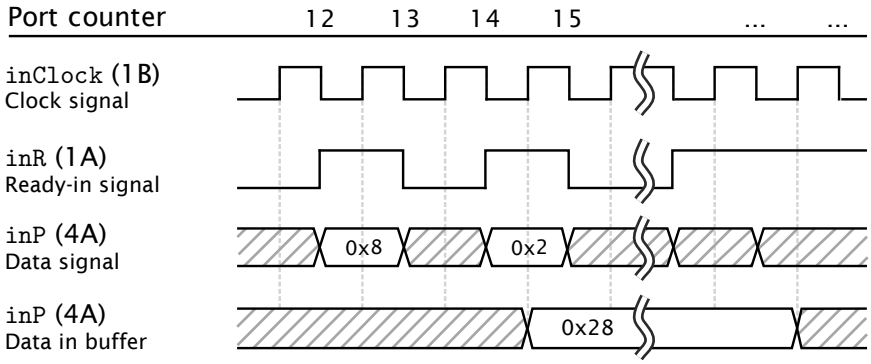
A clocked port can interpret a *ready-in* strobe signal that determines the validity of the accompanying data. The program below inputs data from a clocked port only when a ready-in signal is high.

```
#include <xs1.h>

in buffered port:8 inP     = XS1_PORT_4A;
in           port  inReady = XS1_PORT_1A;
in           port  inClock = XS1_PORT_1B;
clock              clk     = XS1_CLKBLK_1;

int main(void) {
  configure_clock_src(clk, inClock);
  configure_in_port_strobed_slave(inP, inReady, clk);
  start_clock(clk);

  inP :> void;
}
```

The statement

```
configure_in_port_strobed_slave(inP, inReady, clk);
```

configures the input port `inP` to be sampled only when the value sampled on the port `inReady` equals 1. The ready-in port must be 1-bit wide. The waveform diagram below shows example input stimuli and the data input by this program.



Data is sampled on the rising edge of the clock whenever the ready-in signal is high. The port samples two 4-bit values and combines them to produce a single 8-bit value for input by the processor; the data input is 0x28. XS1 devices have a single-entry buffer, which means that data is available for input until the ready-in signal is high for the next two rising edges of the clock. Note that the port counter is incremented on every clock period, regardless of whether the strobe signal is high.

6.4 Outputting Data and a Data Valid Signal

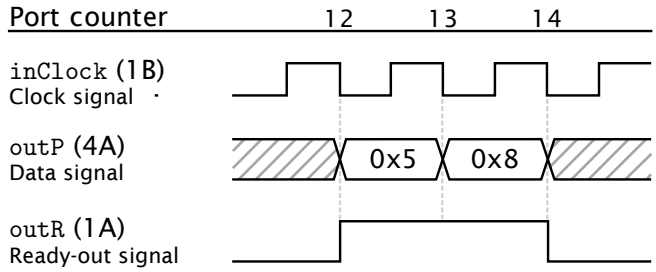
A clocked port can generate a *ready-out* strobe signal whenever data is output. The program below causes an output port to drive a data valid signal whenever data is driven on a 4-bit port.

```
#include <xs1.h>

out buffered port:8 outP    = XS1_PORT_4B;
out           port  outR    = XS1_PORT_1A;
in            port  inClock = XS1_PORT_1B;
clock               clk     = XS1_CLKBLK_1;

int main(void) {
  configure_clock_src(clk, inClock);
  configure_out_port_strobed_master(outP, outR, clk, 0);
  start_clock(clk);

  outP <: 0x85;
}
```

The statement

```
configure_out_port_strobed_master(outP, outR, clk, 0);
```

configures the output port `outP` to drive the port `outR` high whenever data is output. The ready-out port must be 1-bit wide. The waveform diagram below shows the data and strobe signals driven by this program.



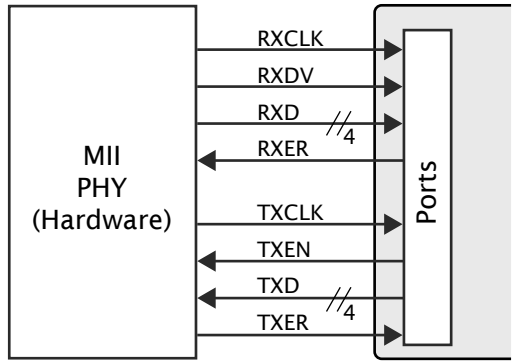
The port drives two 4-bit values over two clock periods, raising the ready-out signal during this time.

It is also possible to implement control flow algorithms that output data using a ready-in strobe signal and that input data using a ready-out strobe signal; when both signals are configured, the port implements a symmetric strobe protocol that uses a clock to handshake the communication of the data (see §C.2.2 and §B.2).

On XS1 devices, ports used for strobing must be qualified with the keyword `buffered`; see §C.1 for further explanation.

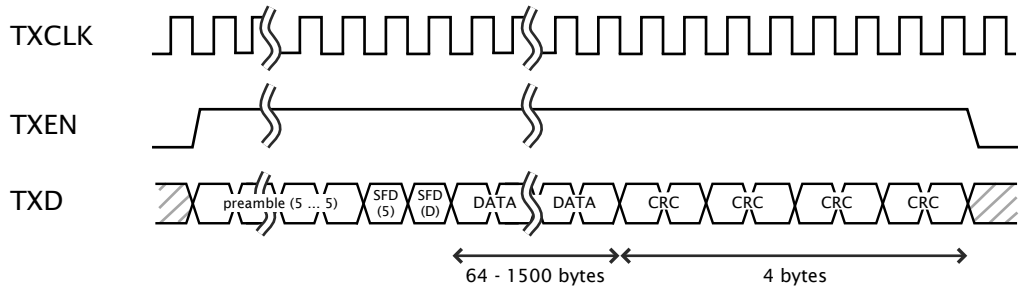
6.5 Case Study: Ethernet MII

A single thread on an XS1 device can be used to implement a full duplex 100Mbps Ethernet Media Independent Interface (MII) protocol [5]. This protocol implements the data transfer signals between the link layer and physical device (PHY). The signals are shown below.



6.5.1 MII Transmit

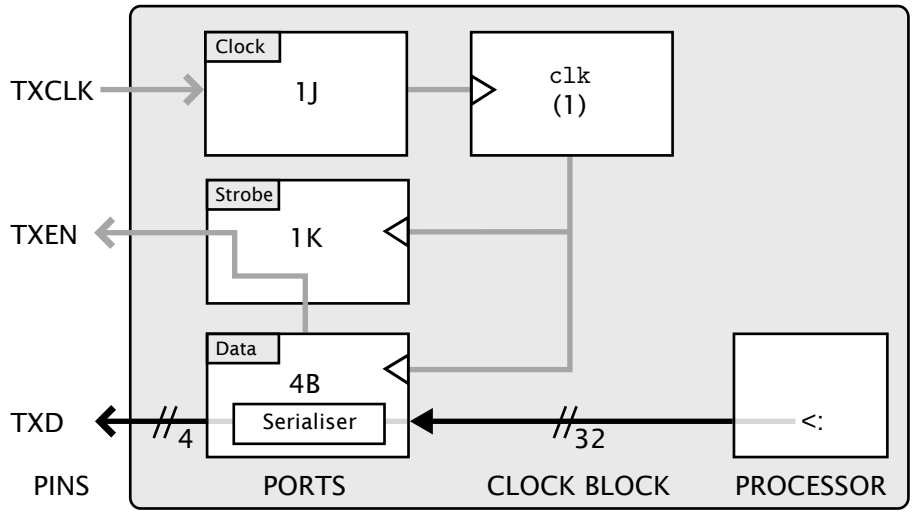
The waveform diagram below shows the transmission of a single frame of data to the PHY. The error signal TXER is omitted for simplicitly.



The signals are as follows:

- TXCLK is a free running 25MHz clock generated by the PHY.
- TXEN is a data valid signal driven high by the transmitter during frame transmission.
- TXD carries a nibble of data per clock period from the transmitter to the PHY. The transmitter starts by sending a preamble of nibbles of value 0x5, followed by two nibbles of values 0x5 and 0xD. The data, which must be in the range of 64 to 1500 bytes, is then transmitted, least significant bit first, followed by four bytes containing a CRC.

The diagram below illustrates the port configuration required to serialise the output data and produce a data valid signal.



The port TXD performs a 32-to-4 bit serialisation of data onto its pins. It is synchronised to the 1-bit port TXCLK and uses the 1-bit port TXEN as a ready-out strobe signal that is driven high whenever data is driven. In this configuration, the processor has only to output data once every eight clock periods and does not need to explicitly output the data valid signal. The program below defines and configures the ports in this way.

```
#include <xs1.h>
out buffered port:32 TXD   = XS1_PORT_4B;
out          port    TXEN  = XS1_PORT_1K;
in           port    TXCLK = XS1_PORT_1J;
clock                clk   = XS1_CLKBLK_1;

void miiConfigTransmit(clock clk,
  buffered out port:32 TXD, out port TXEN) {

  configure_clock_src(clk, TXCLK);
  configure_out_port(TXD, clk);
  configure_out_port(TXEN, clk);
  configure_out_port_strobed_master(TXD, TXEN, clk, 0);
  start_clock(clk);
}
```

The function below inputs frame data from another thread and outputs it to the MII ports. For simplicity, the error signals and CRC are ignored.

```
void miiTransmitFrame(out buffered port:32 TXD,
                      streaming chanend c) {
  int numBytes, tailBytes, tailBits, data;

  /* Input size of next packet */
  c :> numBytes;
  tailBytes = numBytes / 4;
  tailBits = tailBytes * 8;

  /* Output row of 0x5s followed by 0xD */
  TXD <: 0xD5555555;

  /* Output 32-bit words for serialisation */
  for (int i=0; i<numBytes-tailBytes; i+=4) {
    c   :> data;
    TXD <: data;
  }

  /* Output remaining bits of data for serialisation */
  if (tailBits != 0) {
    c :> data;
    partout(TXD, tailBits, data);
  }
}
```

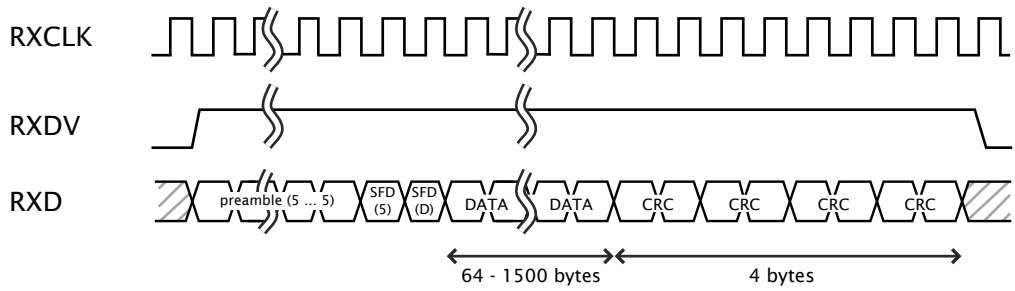
The program first inputs from the channel `c` the size of the frame in bytes. It then outputs a 32-bit preamble to `TXD`, which is driven on the pins as nibbles over eight clock periods. On each iteration of the `for` loop, the next 32 bits of data are then output to `TXD` for serialising onto the pins. This gives the processor enough time to get around the loop before the next block of data must be driven. The final statement

```
partout(TXD, tailBits, data);
```

performs a *partial output* of the remaining bits of `data` that represent valid frame data.

6.5.2 MII Receive

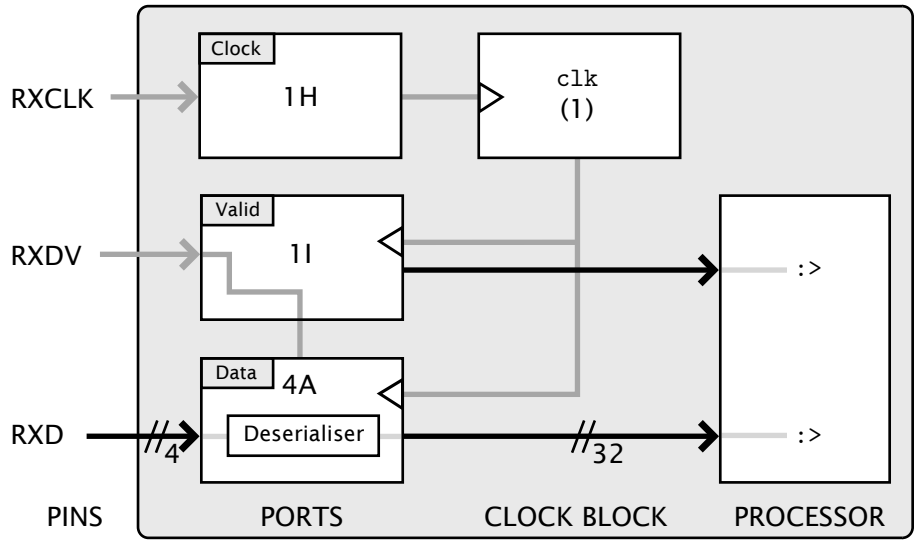
The waveform diagram below shows the reception of a single frame from the PHY. The error signal RXER is omitted for simplicitly.



The signals are as follows:

- RXCLK is a free running clock generated by the PHY.
- RXDV is a data valid signal driven high by the PHY during frame transmission.
- RXD carries a nibble of data per clock period from the PHY to the receiver. The receiver waits for a preamble of nibbles of values 0x5, followed by two nibbles with values 0x5 and 0xD. The actual data is then received, which is in the range of 64 to 1500 bytes, least significant nibble first, followed by four bytes containing a CRC.

The diagram on the next page illustrates the port configuration required to deserialise the input data when a data valid signal is present.



The port RXD performs a 4-to-32-bit deserialisation of data from its pins. It is synchronised to the 1-bit port RXCLK and uses the 1-bit port RXDV as a ready-in strobe signal that causes data to be sampled only when the strobe is high. In this configuration, the port can sample eight values before the data must be input by the processor, and the processor does not need to explicitly wait for the data valid signal. The program below defines and configures the ports in this way.

```
#include <xs1.h>

in buffered port:32 RXD   = XS1_PORT_4A;
in          port    RXDV  = XS1_PORT_1I;
in          port    RXCLK = XS1_PORT_1H;
clock               clk   = XS1_CLKBLK_1;

void miiConfigReceive(clock clk, in port RXCLK,
  buffered in port:32 RXD, in port RXDV, in port RXER) {

  configure_clock_src(clk, RXCLK);
  configure_in_port(RXD, clk);
  configure_in_port(RXDV, clk);
  configure_in_port_strobed_slave(RXD, RXDV, clk);
  start_clock(clk);
}
```

The function on the following page receives a single error-free frame and outputs it to another thread. For simplicity, the error signal and CRC are ignored.

```
#define MORE 0
#define DONE 1

void miiReceiveFrame(in buffered port:32 RXD, in port RXDV,
                     streaming chanend c) {
  int notDone = 1;
  int data, tail;

  /* Wait for start of frame */
  RXD when pinseq(0xD) :> void;

  /* Receive frame data/crc */
  do {
    select {
      case RXD :> data :
        /* input next 32 bits of data */
        c <: MORE;
        c <: data;
        break;
      case RXDV when pinseq(0) :> notDone :
        /* Input any bits remaining in port */
        tail = endin(RXD);
        for (int byte=tail>>3; byte > 0; byte-=4) {
          RXD :> data;
          c <: MORE;
          c <: data;
        }
        c <: DONE;
        c <: tail >> 3;
        break;
    }
  } while (notDone);
}
```

The processor waits for the last nibble of the preamble (0xD) to be sampled by the port RXD. Then on each iteration of the loop, it waits for either next eight nibbles of data to be sampled for input by RXD or for the data valid signal RXDV to go low.

An effect of using a port's serialisation and strobing capabilities together is that the ready-in signal may go low before a full transfer width's worth of data is received. The statement

```
tail = endin(RXD);
```

causes the port RXD to respond with the remaining number of bits not yet input. It also causes the port to provide this data on the subsequent inputs, even though the data valid signal is low and the shift register is not yet full.

XS1 devices provide a single-entry buffer up to 32-bits wide and a 32-bit shift register, requiring up to 64 bits of data being input over two input statements once the data valid signal goes low.

6.6 Summary

The semantics for I/O on a serialised port are as follows (where p refers to the port width and w refers to the transfer width of a port):

- An output of a w-bit value is driven over $\frac{w}{p}$ consecutive clock periods, least significant bits first. The ready-out signal is driven high on each of these periods.
- For a timed output, the port waits until its counter equals the specified time before *starting* to serialise the data. The ready-out signal is not driven while waiting to serialise.
- An input of a w-bit value is sampled over $\frac{w}{p}$ clock periods, with earlier bits received ending up in the least significant bits of w. (If a ready-in signal is used, the clock periods may not be consecutive.)
- For a timed input, the port provides the *last* p bits of data sampled when its counter equals the specified time.

If a port is configured with a ready-in signal:

- Data is sampled only on rising edges of the port's clock when the ready-in signal is high.

If a port is configured with a ready-out signal:

- The ready-out signal is driven high along with the data and is held for a single period of the clock.

A full description of the semantics for strobing and serialisation is given in Appendix B.

APPENDIX A

XC Language Specification

The specification given in this appendix describes version 9.9 of XC; the behaviour of I/O operations on ports is given separately in Appendix B.

> The layout of this manual and portions of its text are based upon the K&R definition of C [6]. Commentary material highlighting differences between XC and C is indented and written in smaller type.

A.1 Lexical Conventions

A program consists of one or more *translation units* stored in files. It is translated in several phases, which are described in §A.13. The first phases perform low-level lexical transformations, carry out directives introduced by lines beginning with the # character, and perform macro definition and expansion. When the preprocessing of §A.13 is complete, the program has been reduced to a sequence of tokens.

A.1.1 Tokens

There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. Blank spaces, horizontal tabs, newlines, formfeeds, and comments as described below, collectively referred to as *white space*, are ignored except as they separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords and constants.

A.1.2 Comments

Two styles of commenting are supported: the characters `/*` introduce a comment, which terminates with the characters `*/`, and the characters `//` introduce a comment, which terminates with a newline. Comments may not be nested, and they may not occur within string or character literals.

A.1.3 Identifiers

An identifier is a sequence of letters, digits and underscore (_) characters of any length; the first character must not be a digit. Upper and lower case letters are different.

A.1.4 Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

```
auto        else        return      union
break       enum        short       unsigned
case        extern      signed      void
char        for         sizeof      volatile
const       if          static      while
continue    int         struct
default     long        switch
do          register    typedef
```

The following identifiers are also reserved for use as keywords and may not be used otherwise:

```
buffered    inline      out         slave
chan        isnull      par         streaming
chanend     master      port        timer
core        null        select      transaction
in          on          service     when
```

The construction `port:`*n* where *n* is a sequence of digits is also a valid identifier. The sequence of digits is taken to be decimal and is interpreted as an integer constant. The following identifiers are reserved for compatibility issues and for future use:

```
accept      claim       float       restrict
asm         double      module
```

A.1.5 Constants

There are several kinds of constants. Each has a data type; §A.3.2 discusses the basic types.

constant ::= *integer-constant*
| *character-constant*
| *enumeration-constant*
| `null`

Floating-point constants are unsupported.

A.1.5.1 Integer Constants

A sequence of digits is taken to be binary if preceded by `0b` or `0B`, octal if preceded by `0`, hexadecimal if preceded by `0x` or `0X`, and decimal otherwise. integer constant may be suffixed by the letter `u` or `U` (unsigned), the letter `l` or `L` (long), or both (unsigned long).

The type of an integer constant depends on its form, value and suffix. (See §A.3 for a discussion of types.) An unsuffixed decimal constant has the first of the following types in which its value can be represented: `int`, `long int`, `unsigned long int`; an unsuffixed octal or hexadecimal constant has the first possible of types: `int`, `unsigned int`, `long int`, `unsigned long int`. An unsigned constant has the first possible of types: `unsigned int`, `unsigned long int`; a long constant has the first possible of types: `long int`, `unsigned long int`.

A.1.5.2 Character Constants

A character constant is a sequence of one or more characters (excluding the single-quote and newline characters) enclosed in single quotes. The value of a character constant with a single character is the numeric value of the character in the machine's character set at execution time. The value of a multi-character constant is implementation-defined.

Wide character constants are unsupported.

The following escape sequences are supported.

newline	NL	`\n`	backslash	`\`	`\\`
horizontal tab	HT	`\t`	question mark	`?`	`\?`
vertical tab	VT	`\v`	single quote	`'`	`\'`
backspace	BS	`\b`	double quote	`"`	`\"`
carriage return	CR	`\r`	octal number	*ooo*	`\`*ooo*
formfeed	FF	`\f`	hex number	*hh*	`\x`*hh*
audible alert	BEL	`\a`			

The escape sequence `\`*ooo* requires one, two or three octal digits. The sequence `\x`*hh* requires one or more hexadecimal digits; its behaviour is undefined if the resulting character value exceeds that of the largest character. For either octal or hexadecimal escape characters, if the implementation treats the `char` type as signed, the value is sign-extended as if cast to `char` type. If any other character follows the \ then the behaviour is undefined.

A.1.5.3 Enumeration Constants

Identifiers declared as enumerators (see §A.7.5) are constants of type `int`.

A.1.5.4 Null Constants

The null constant has type `null`.

A.1.6 String Literals

A string literal is a sequence of zero or more characters (excluding the double-quote and newline characters) enclosed in double quotes. It has type "array of characters" and storage class `static` (see §A.3.1) initialised with the given characters. Whether identical string literals are distinct is implementation-defined, and the behaviour of a program that attempts to alter a string literal is undefined.

Adjacent string literals are concatenated into a single string. After any concatenation, a null byte `\0` is appended to the string. All of the character escape sequences are supported.

A.2 Syntax Notation

In the syntax notation used in this manual, syntactic categories are indicated by serif type, and literal words and characters by `typewriter` style. An optional terminal or nonterminal symbol carries the subscripted suffix "*opt*," so that, for example,

{ expression$_{opt}$ }

means an optional expression, enclosed in braces. The terms "zero or more" and "one or more" are represented using angled brackets along with the star (*) and plus (+) symbols

respectively, so that, for example,

⟨declaration⟩*

means a sequence of zero or more declarations, and

⟨declaration⟩+

means a sequence of one or more declarations.

A.3 Meaning of Identifiers

Identifiers (or names) refer collectively to functions, tags of structures and unions, members of structures or unions, and objects. An object (or variable) is a location in storage, and its interpretation depends on its *storage class* and its *type*. The storage class determines the lifetime of the storage associated with the identifier; the type determines the meaning of the values found in the identified object. A name also has scope, which is the region of the program in which it is known, and a linkage, which determines whether the same name in another scope refers to the same object or function. Scope and linkage are discussed in §A.10.

A.3.1 Storage Class

An object has either *automatic* or *static* storage. Automatic objects are local to a block (§A.8.4) and are discarded on exit from the block. Declarations within a block create automatic objects if no storage class is mentioned, or if the `auto` or `register` specifier is used.

Static objects may be local to a block or external to all blocks, but in either case retain their values across exit from and reentry to functions and blocks. Within a block, static objects are declared with the keyword `static`. The objects declared outside all blocks, at the same level as function definitions, are always static. They may be made local to a particular translation unit by use of the `static` keyword; this gives them *file-scope* (or *internal linkage*). They become global to an entire program by omitting an explicit storage class, or by using the keyword `extern`; this gives them *program-scope* (or *external linkage*).

A function may be declared with the keyword `service`. This specifier has no effect on the behaviour of the function; the extent to which suggestions made by using this specifier are effective is implementation-defined.

A.3.2 Basic Types

Objects declared as `char` are large enough to store any member of the execution character set. If a genuine character from that set is stored in a `char` object, its value is equivalent to the integer code for the character, and is non-negative. Other quantities may be stored into `char` variables, but the available range of values, and especially whether the value is signed, is implementation-defined.

Objects declared `unsigned char` consume the same amount of space as plain characters, but always appear non-negative; explicitly signed characters declared `signed char` likewise take the same space as plain characters.

In addition to the `char` type, up to three sizes of integer are available, declared `short int`, `int` and `long int`. Plain `int` objects have the natural size suggested by the host machine architecture. Longer integers provide at least as much storage as shorter ones, but the

implementation may make plain integers equivalent to either short or long integers. The `int` types all represent signed values unless specified otherwise.

Unsigned integers obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation. The set of non-negative values that can be stored in a signed object is a subset of the values that can be stored in the corresponding unsigned object, and the representation for the overlapping values is the same.

All of the above types are collectively referred to as *arithmetic* types, because they can be interpreted as numbers, and as *integral* types, because they represent integer values.

The `void` type specifies an empty set of values; it is used as the type returned by functions that generate no value.

> The C types `long long int`, `float` and `double` are unsupported.

The `chan` type specifies a logical communication channel over which values can be communicated between parallel statements (§A.8.8). The `chanend` type specifies one end of a communication channel.

The locations of at most two implied ends of `chan` (themselves `chanends`) are defined through the use of the channel in at most two parallel statements (§A.8.8).

Channel ends are used as operands of input and output statements (§A.8.3). Channels are bidirectional and synchronised: an outputter waits for a matching inputter to become ready before data is communicated. Whether a streaming channel is synchronised or unsynchronised is implementation-defined.

The `port` type specifies a *p*-bit register, which interfaces to a collection of *p* pins used for communicating with the environment where *p* is implementation-defined. The `port:`*n* type specifies an *n*-bit register, which interfaces to a collection of *p* pins used for communicating with the environment (where *p* need not equal *n*). A `void port` is a special type of port that may not be used for input or output. A port also has a notional *transfer* type and *counter* type (see §A.8.3); these types are implementation-defined.

Ports are used as operands of input and output statements (§A.8.3, Appendix B).

The `timer` type is a special type of input port that returns the current time when input from. A `void timer` is a timer that may not used for input. A timer also has a notional *counter* type (see §A.8.3); this type is implementation-defined.

The `core` type specifies a processor core on which ports and parallel statements may be placed. Objects of core type do not reserve storage.

Channel ends, ports, timers and cores are collectively referred to as having *resource* types. Except for cores, which do not reserve storage, an object of resource type refers to a location in storage in which an identifier for the resource is recorded.

> `chan`, `chanend`, `port`, `timer`, `core` and `buffered` and `streaming` are new.

A.3.3 Derived Types

In addition to the basic types, the following derived types may be constructed in the following ways:

- *Arrays* of objects of a given type.
- *Functions* returning objects of a given type.
- *References* to objects of a given type.
- *Structures* containing a sequence of objects of various types.

- *Unions* capable of containing any one of several objects of various types.
- *Lists* of objects containing a sequence of objects of various types.

In general these methods of constructing objects can be applied recursively.

> Lists of types are used in multiple assignment statements (§A.8.2); pointers are replaced by references (see §A.6.1; §A.6.3.2; §A.7.7.2).

A.3.4 Type Qualifiers

An object's type may be qualified `const`, which announces that its value will not be changed; its range of values and arithmetic properties is unaffected.

A port may be qualified `in` or `out`, which announces that it will only be used for input or output operators (§A.8.3).

Qualifiers are discussed in §A.6.3.2 and §A.7.2.

> `in` and `out` are new.

A.4 Objects and Lvalues

An *object* is a named region of storage; an *lvalue* is a reference to an object. For example, if `str` is an identifier of type "1-dimensional array of char" then `str[0]` is an lvalue referring to the character object indexed by the first element of the array `str`.

A *modifiable lvalue* is an lvalue which is modifiable: it must not be an array, and must not have a resource or incomplete type, or be a function. Also, its type must not be qualified with `const`; if it is a structure or union, it must not have any member or, recursively, submember qualified with `const`.

A.5 Conversions

Some operators, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the results to be expected from such conversions. §A.5 details the conversions demanded by most operators.

A.5.1 Integral Promotion

A character or a short integer, both either signed or not, may be used in an expression wherever an integer may be used. If an `int` can represent all the values of the original type then the value is converted to `int`, otherwise the value is converted to `unsigned int`.

A.5.2 Integral Conversions

Any integer is converted to a given unsigned type by finding the smallest non-negative value that is congruent to that integer, modulo one more than the largest value that can be represented in the unsigned type.

When any integer is converted to a signed type, its value is unchanged if it can be represented in the new type, and implementation-defined otherwise.

A.5.3 Arithmetic Conversions

Many operators cause conversions which bring their operands into a common type, which is also the type of the result. The rules for performing these *usual arithmetic conversions* are as follows:

- First, integral promotions are performed on both operands.
- If either operand is `unsigned long int` then the other is converted to `unsigned long int`.
- Otherwise, if one operand is `long int` and the other is `unsigned int` then: if a `long int` can represent all values of an `unsigned int` then the `unsigned int` operand is converted to `long int`, otherwise both operands are converted to `unsigned long int`.
- Otherwise, if one operand is `long int` then the other is converted to `long int`.
- Otherwise, if either operand is `unsigned int` then the other is converted to `unsigned int`.
- Otherwise both operands have type `int`.

A.5.4 Void

The (nonexistent) value of a `void` object may not be used in any way, and neither explicit nor implicit conversion to any non-void type may be applied. An object of type `void port` or `void timer` may not be used for input or output.

A.6 Expressions

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Within each section, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein.

The precedence and associativity of operators is fully specified. The order of evaluation of expressions does not, with certain exceptions, affect the behaviour of the program, even if the subexpressions involve side effects. In particular, a variable which is changed in one part of an expression may not unless otherwise stated appear in any other part of the expression. This rule applies recursively to all variables which are changed in functions called in the expression.

The handling of overflow, divide check, and other exceptions in expression evaluation is implementation-defined.

A.6.1 Reference Generation

If the type of an expression is “array of *T*,” for some type *T*, then the value of the expression is a reference to the array, and the type of the expression is altered to “reference to *T*.”

A.6.2 Primary Expressions

Primary expressions are variable references, function calls, constants, strings, or expressions in parentheses:

primary-expression ::= *variable-reference*
| *function-call*
| *constant*
| *string*
| (*expression*)

A variable reference is a primary expression, providing the identifier named (§A.6.3) has been suitably declared as discussed below; the type of the identifier is specified by its declaration; the type of the expression is that of the identifier.

A function call is a primary expression; the type the expression depends on the return type of the function (§A.6.3.2).

A constant is a primary expression; the type of the expression is that of the constant (which depends on its form discussed in §A.1.5).

A string literal is a primary expression; the type of the expression is "array of `char`."

A parenthesised expression is a primary expression whose type and value are identical to those of the unadorned expression.

A.6.3 Postfix Expressions

The operators in postfix expressions group left to right.

postfix-expression ::= *primary-expression*
| *variable-reference* ++
| *variable-reference* --

variable-reference ::= *identifier*
| *variable-reference* [*expression*]
| *variable-reference* . *identifier*
| (*variable-reference* , *type-name*)

function-call ::= *identifier* (*expression-list*$_{opt}$)

expression-list ::= *expression*
| *expression* , *expression-list*

A.6.3.1 Array References

A variable reference followed by an expression in square brackets is a subscripted array reference. The variable reference must either have type "*n*-array of *T*" or "reference to an *n*-array of *T*," where *n* is the number of dimensions and *T* is some type, and the expression must have integral type; the type of the subscripted variable reference is *T*. If the value of the expression is less than zero or greater than or equal to *n* then the expression is invalid. See §A.7.7.1 for further discussion.

A.6.3.2 Function Calls

A function call is an identifier followed by parentheses containing an optional list of comma-separated expressions, which constitute the arguments to the function. If the identifier has type "transaction function returning `void`" then the call must be within the scope of a transaction statement (§A.8.9). Otherwise, the identifier must have type "function returning *T*," or "select function returning *T*," for some type *T*, in which case the value of the function call has type *T*.

> Function declarations are limited to file-scope only (§A.9). Implicit function declarations (see K&R §A7.3.2) are unsupported.

The term *argument* refers to an expression passed by a function call, and the term *parameter* refers to an input object (or its identifier) received by a function definition, or described in a function declaration.

If the type of a parameter is "reference to *T*," for some *T*, then its argument is passed by reference, otherwise the argument is passed by value. In preparing for the call to a function, a copy is made of each argument that is passed by value. A function may change the values of these parameter objects, which are copies of the argument expressions, but these changes cannot affect the values of the arguments. For objects that are passed by reference, a function may change the values that these objects dereference, thereby affecting the values of the arguments. For the purpose of disjointness checking, passing an object by reference counts as a write to that object unless the type of the parameter is qualified as `const` or an array of objects qualified as `const`.

For arguments passed by value, the argument and parameter are deemed to agree in type if the promoted type of the argument is that of the parameter itself, without promotion. For arguments passed by reference, the argument and parameter agree in type only if the types are equivalent (see A.7.11) ignoring all qualifiers and array sizes, and obey the following rules:

- An object or an array of objects declared with the qualifier `const` may only be used as an argument to a function with parameter qualified `const`.
- An object declared with the qualifier `in` may only be used as an argument to a function with parameter qualified `in` or `void`.
- An object declared with the qualifier `out` may only be used as an argument to a function with parameter qualified `out` or `void`.
- An object declared with the specifier `void` may only be used as an argument to a function with parameter specified `void`. An object not qualified `in`, `out` or `void` may be used as an argument to a function with parameter qualified either `in`, `out` or `void`.
- An object declared with the qualifier `buffered` may only be used as an argument to a function parameter qualified `buffered`. An object not declared with the qualifier `buffered` may only be used as an argument to a function parameter not qualified `buffered`.
- An object declared with the qualifier `streaming` may only be used as an argument to a function parameter qualified `streaming`. An object not declared with the qualifier `streaming` may only be used as an argument to a function parameter not qualified `streaming`.

- An object declared with an array size of *n* may only be used as an argument to a function parameter that is an array of unknown size or of size *m* where *m* <= *n*.
- An object passed to a parameter declared without the qualifier `const` must be an lvalue.

A variable which is changed in one argument may not appear in any other argument. This rule applies recursively to all variables appearing in functions called by the arguments.

The arguments passed by value are converted, as if by assignment, to the types of the corresponding parameters of the function's declaration (or prototype). The number of arguments must be the same as the number of parameters, unless the declaration's parameter list ends with the ellipsis notation (`, ...`). In that case, the number of arguments must equal or exceed the number of parameters; trailing integral arguments beyond the explicitly typed parameters undergo integral promotion (§A.5.1).

The order of evaluation of arguments is unspecified, but the arguments are completely evaluated, including all side effects, before the function is entered. Recursive calls to any function are permitted.

The creation of more than one reference to the same object of basic type, a structure, a union or an array is invalid. The creation of a reference to a structure, union or array, and to a member or element recursively contained within is invalid. The creation of more than one reference to objects contained within distinct members of a union is invalid.

A.6.3.3 Structure References

A variable reference followed by a dot followed by an identifier is a member reference. The variable reference must be a structure or union, and the identifier must name a member of the structure or union. The value is the named member of the structure or union, and its type is the type of the member.

Structures and unions are discussed in §A.7.4.

A.6.3.4 Reinterpretation

A left parenthesis followed by a variable reference followed by a comma followed by a type name (§A.7.9) followed by a right parenthesis is a reinterpretation cast.

The variable reference must not specify a resource type; its type must be complete or it must be an incomplete array with the first dimension missing which, if provided, completes the type. The variable type name must not be a resource type; it must be complete.

If the size of the type of the variable reference is unknown because it references an array parameter with unknown size then the following two rules apply. First, if the size of the type name is a compile-time constant *T* then at run-time if the size of the variable reference is less than *T* then the reinterpret operation is invalid. Second, if the size of the type name is not known at compile-time because it is an array in which the largest dimension is unspecified then at run-time the reinterpret operation provides a value for the dimension *d* such that the size of the resulting type is not larger than the size of the type of the variable reference, but with a value of *d+1* it would be.

If the size of the type of the variable reference is a compile-time constant *V* then the following two rules apply. First, if the size of the type name is a compile-time constant *T* then *T* must not be greater than *V*. Second, if the size of the type name is unknown because it references an array in which the largest dimension is unspecified then a value for this

dimension *d* is completed at compile-time such that the size of the resulting type is not larger than *V*, but with a value of *d+1* it would be.

No arithmetic conversions are performed: the effect of the reinterpretation is to treat the variable as if it had the specified type. An array of size zero is a valid reinterpretation; any attempted index into the array is invalid.

The use of a reinterpreted object may be invalid if it is not suitably aligned in storage. It is guaranteed only that an object may be reinterpreted to an object whose type requires less or equally strict storage alignment; the notion of "alignment" is implementation-defined, but objects of the `char` types have least strict alignment requirements.

A.6.4 Unary Operators

Expressions with unary operators group right-to-left.

unary-expression ::= *postfix-expression*
| `++` *variable-reference*
| `--` *variable-reference*
| *unary-operator cast-expression*
| `sizeof` *unary-expression*
| `sizeof (` *type-name* `)`
| `isnull (` *unary-expression* `)`

unary-operator ::= *one of*
`+ - ~ !`

A.6.4.1 Prefix Incrementation Operators

A unary expression preceded by a ++ or -- operator is a unary expression. The operand is incremented (++) or decremented (--) by 1. The value of the expression is the value after the incrementation (decrementation). The operand must be a modifiable lvalue; see the discussion of additive operators (§A.6.7) and assignment (§A.6.17) for details of the operation. The result is not an lvalue.

A.6.4.2 Unary Plus Operator

The operand of the unary + operator must have arithmetic type, and the result is the value of its operand. An integral operand undergoes integral promotion; the type of the result is the type of the promoted operand.

A.6.4.3 Unary Minus Operator

The operand of the unary - operator must have arithmetic type, and the result is the negative of its operand. An integral operand undergoes integral promotion. The negative of an unsigned quantity is computed by subtracting the promoted value from the largest value of the promoted type and adding one; but negative zero is zero. The type of the result is the type of the promoted operand.

A.6.4.4 One's (Bitwise) Complement Operator

The operand of the unary ~ operator must have integral type, and the result is the one's complement of its operand. The integral promotions are performed. If the operand is unsigned, the result is computed by subtracting the value from the largest value of the promoted type. If the operand is signed, the result is computed by converting the promoted operand to the corresponding unsigned type, applying ~, and converting back to the signed type. The type of the result is the type of the promoted operand.

A.6.4.5 Logical Negation Operator

The operand of the `!` operator must have arithmetic type, and the result is 1 if the value of its operand compares equal to 0, and 0 otherwise. The type of the result is `int`.

A.6.4.6 Sizeof Operator

The `sizeof` operator yields the number of bytes required to store an object of the type of its operand. The operand is either an expression, which is not evaluated, or a parenthesised type name. When `sizeof` is applied to a `char`, the result is 1; when applied to an array, the result is the total number of bytes in the array. When applied to a structure or union, the result is the number of bytes in the object, including any padding required to make the object tile an array: the size of an array of *n* elements is *n* times the size of one element. When applied to a reference, the result is the number of bytes in the object referred to. The operator may not be applied to an operand of function type, of resource type or of an incomplete type. The operator may not be applied to an operand of reference type where the reference is to an array of unknown size. The value of the result is implementation-defined. The result is an unsigned integral constant; the particular type is implementation-defined.

A.6.4.7 Isnull Operator

The operand of the `isnull` operator must be an lvalue. The result is 1 if its operand has value `null`, and 0 otherwise. The type of the result is `int`.

A.6.5 Casts

A unary expression preceded by the parenthesised name of a type causes conversion of the value of the expression to the named type.

cast-expression ::= *unary-expression*
| (*type-name*) *cast-expression*

This construction is called a *cast*. The cast must not specify a structure, a union, an array, or a resource type; neither must the expression. Type names are described in §A.7.9. The effects of arithmetic conversions are described in §A.5.3. An expression with a cast is not an lvalue.

A.6.6 Multiplicative Operators

The multiplicative operators *, / and % group left-to-right.

multiplicative-expression ::= *cast-expression*
| *multiplicative-expression* * *cast-expression*
| *multiplicative-expression* / *cast-expression*
| *multiplicative-expression* % *cast-expression*

The operands of * and / must have arithmetic type; the operands of % must have integral type. The usual arithmetic conversions are performed on the operands, and determine the type of the result.

The binary * operator denotes multiplication.

The binary / operator produces the quotient, and the % operator the remainder, of the division of the first operand by the second; if the second operand is 0 then the result is implementation-defined. Otherwise, it is always true that `(a/b)*b + a%b` is equal to `a`. If both operands are non-negative, then the remainder is non-negative and smaller than the divisor; if not, it is guaranteed only that the absolute value of the remainder is smaller than the absolute value of the divisor.

A.6.7 Additive Operators

The additive operators + and - group left-to-right.

additive-expression ::= *multiplicative-expression*
| *additive-expression* + *multiplicative-expression*
| *additive-expression* - *multiplicative-expression*

For both operators, each operand must have arithmetic type. The usual arithmetic conversions are performed on the operands, and determine the type of the result.

The result of the + operator is the sum of the operands, and the result of the - operator is the difference of the operands.

A.6.8 Shift Operators

The shift operators << and >> group left-to-right. For both operators, each operand must be integral, and is subject to integral promotions. The type of the result is that of the promoted left operand. The result is undefined if the right operand is negative, or greater than or equal to the number of bits in the left expression's type.

shift-expression ::= *additive-expression*
| *shift-expression* << *additive-expression*
| *shift-expression* >> *additive-expression*

The result of the << operator is the left operand left-shifted by the number of bits specified by the right operand. The value of the >> operator is the left operand right-shifted by the number of bits specified by the right operand.

A.6.9 Relational Operators

The relational operators < (less), > (greater), <= (less or equal) and >= (greater or equal) group left-to-right (but this fact is not useful).

relational-expression ::= *shift-expression*
| *relational-expression* < *shift-expression*
| *relational-expression* > *shift-expression*
| *relational-expression* <= *shift-expression*
| *relational-expression* >= *shift-expression*

For all of these operators, each operand must have arithmetic type. The usual arithmetic conversions are performed; the type of the result is `int`.

All of these operators produce a result of 0 if the specified relation is false and 1 if it is true.

A.6.10 Equality Operators

equality-expression ::= *relational-expression*
| *equality-expression* == *relational-expression*
| *equality-expression* != *relational-expression*

The equality operators == (equal to) and != (not equal to) are analogous to the relational operators except for their lower precedence.

A.6.11 Bitwise AND Operator

AND-expression ::= *equality-expression*
| *AND-expression* & *equality-expression*

The operands of the bitwise AND operator & must have integral type. The usual arithmetic conversions are performed; the result is the bitwise AND function of the operands.

A.6.12 Bitwise Exclusive OR Operator

exclusive-OR-expression ::= *AND-expression*
| *exclusive-OR-expression* ^ *AND-expression*

The operands of the bitwise exclusive OR operator ^ must have integral type. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of its operands.

A.6.13 Bitwise Inclusive OR Operator

inclusive-OR-expression ::= *exclusive-OR-expression*
| *inclusive-OR-expression* | *exclusive-OR-expression*

The operands of the bitwise inclusive OR operator | must have integral type. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands.

A.6.14 Logical AND Operator

logical-AND-expression ::= *inclusive-OR-expression*
| *logical-AND-expression* `&&` *inclusive-OR-expression*

The logical AND operator `&&` groups left-to-right. It returns 1 if both its operands compare unequal to zero, 0 otherwise. It guarantees left-to-right *short-circuit* evaluation: the right operand is evaluated only if the left operand evaluates to 1. The operands must have arithmetic type, but need not be the same type; the type of the result is `int`. A variable which is changed by one operand may appear in the other operand.

A.6.15 Logical OR Operator

logical-OR-expression ::= *logical-AND-expression*
| *logical-OR-expression* `||` *logical-AND-expression*

The logical OR operator `||` groups left-to-right. It returns 1 if either of its operands compares unequal to zero, 0 otherwise. It guarantees left-to-right *short-circuit*: the right operand is evaluated only if the left operand evaluates to 0. The operands must have arithmetic type, but need not be the same type; the type of the result is `int`. A variable which is changed by one operand may appear in the other operand.

A.6.16 Conditional Operator

conditional-expression ::= *logical-OR-expression*
| *logical-OR-expression* `?`
expression `:` *conditional-expression*

If the neither the second and third operands have `null` type they must have equivalent types (see A.7.11) ignoring all qualifiers except for `buffered` and `streaming`, and any array sizes, or they must both have arithmetic type.

The first expression is evaluated including all side effects; if it compares unequal to 0, the result is the value of the second expression, otherwise the result is the value of the third expression. If either the second or third operand has type `null`, the result has the type of the other operand. Otherwise, if the second and third operands have equivalent types ignoring qualifiers and any array sizes, the result type has the common type with qualifiers determined by the following rules:

- If either operand is qualified `const`, the result is qualified `const`.
- If either operand is specified with `void`, the result is specified with `void`.
- If one operand is qualified `in` and the other operand is qualified `out`, the result is specified with `void`. Otherwise, if either operand is qualified `in` or `out`, the result is also qualified `in` or `out` respectively. If the operands are arrays of different sizes, the result has type “array of unknown size.”

If the second and third operands have arithmetic type but are not equivalent, the usual arithmetic conversions are performed, and determine the type of the result.

The expression is a lvalue if no arithmetic conversions are performed and the second and third operands both have type `null` or are lvalues.

A.6.17 Assignment Expressions

There are several assignment operators; all group right-to-left.

assignment-expression ::= *conditional-expression*
| *variable-reference assignment-operator*
assignment-expression

assignment-operator ::= *one of*
`= *= /= %= += -= <<= >>= &= ^= |=`

All require a modifiable lvalue as the left operand. The identifier named by the *variable-reference* may appear in any other parts of the assignment, including recursively any functions called, as long as the variables named by the identifiers in these parts are not changed. The type of an assignment expression is that of its left operand, and the value is the value stored in the left operand after the assignment has taken place.

In the simple assignment with `=`, the value of the expression replaces that of the object referred to by the lvalue. One of the following must be true: both operands have arithmetic type, in which case the right operand is converted to the type of the left by the assignment; or both operands are structures or unions of the same type.

An expression of the form `V` *`op`*`=` `E` is equivalent to `V = V` *`op`* `(E)` except that `V` is evaluated only once.

A.6.18 Comma Operator

A restricted form of the comma operator is supported in `for` loops (see §A.8.6).

A.6.19 Constant Expressions

Syntactically, a constant expression is an expression restricted to a subset of operators.

constant-expression ::= *conditional-expression*

Expressions that evaluate to a constant are required in several contexts: after `case` in labelled statements, as array bounds, and in certain preprocessor expressions.

Constant expressions may not contain assignments, increment or decrement operators or function calls, except in an operand of `sizeof`. If the constant expression is required to be integral, its operands must consist of integer, enumeration and character constants; casts must specify an integral type.

A.7 Declarations

Declarations specify the interpretation given to each identifier. Declarations that reserve storage are called *definitions*. The syntax of declarations is:

declaration ::= *on-statement*$_{opt}$ *actual-declaration*

actual-declaration ::= *var-declaration*
| *fnc-declaration* ;
| *trn-declaration* ;
| *sel-declaration* ;

on-statement ::= `on` *variable-reference* `:`

var-declaration ::= ⟨*dec-specifier*⟩* *init-var-declarator-list*$_{opt}$ `;`

fnc-declaration ::= ⟨*dec-specifier*⟩* *fnc-declarator*
| `{` *dec-specifier-list* `}` *fnc-declarator*

trn-declaration ::= ⟨*dec-specifier*⟩* `transaction` *fnc-declarator*

sel-declaration ::= ⟨*dec-specifier*⟩* `select` *fnc-declarator*

dec-specifier-list ::= ⟨*dec-specifier*⟩*
| *dec-specifier-list* `,` ⟨*dec-specifier*⟩*

A variable declaration prefixed with `on` must declare an object of type `port` or `port:`*n*. The variable following `on` must refer to an object of type `core`.

`on` does not change the meaning of the declaration it prefixes.

The *var-declarator*s in the *init-var-declarator-list* and the *fnc-declarator* (see §A.7.6) contain the identifiers being declared.

dec-specifier ::= *storage-class-specifier*
| *type-specifier*
| *type-qualifier*
| `inline`

init-var-declarator-list ::= *init-var-declarator*
| *init-var-declarator* `,` *init-var-declarator-list*

init-var-declarator ::= *var-declarator* ⟨`=` *initialiser*⟩$_{opt}$

Declarators are discussed later (§A.7.6); they contain the names being declared. A declaration must have at least one declarator, or its type specifier must declare a structure tag or a union tag; empty declarations are not permitted.

A.7.1 Storage Class Specifiers

The storage class specifiers are:

storage-class-specifier ::= `auto`
| `register`
| `static`
| `extern`
| `typedef`
| `service`

The meanings of the storage classes were discussed in §A.3.

The `auto` and `register` specifiers give the declared objects automatic storage class, and may be used only within functions. Such declarations also serve as definitions and cause storage to be reserved.

The `static` specifier gives the declared objects static storage class, and may be used either inside or outside functions. Inside a function, this specifier causes storage to be allocated, and serves as a definition; for its effect outside a function, see §A.10.2.

The `extern` specifier, used inside a function, specifies that the storage for the declared objects is defined elsewhere; for its its effects outside a function see §A.10.2.

The `typedef` specifier does not reserve storage and is called a storage class specifier for syntactic convenience; it is discussed in §A.7.10.

At most one of each of the storage class specifiers may be given in a declaration. If none is given, these rules are used: objects declared inside a function are taken to be `auto`; objects and functions declared outside a function, at file-scope, are taken to be static, with external linkage. The specifier `service` may only be given with external function declarations.

> The use of `extern` inside a function is unsupported.

A.7.2 Type Specifiers

The type specifiers are:

type-specifier ::= `void`
| `char`
| `short`
| `int`
| `long`
| `signed`
| `unsigned`
| `chan`
| `chanend`
| `port`
| `port:`*n*
| `timer`
| `core`
| *struct-or-union-specifier*
| *enum-specifier*
| *typedef-name*

At most one of `long` or `short` may be specified together with `int`; the meaning is the same if `int` is not specified. At most one of `signed` or `unsigned` may be specified together with `int`, `short`, `long` or `char`; either may appear alone, in which case `int` is understood. The `signed` specifier is useful for forcing `char` objects to carry a sign; it is permissible but redundant with other integral types. `void` may be specified together with `port` or `port:`*n* to declare a void port; it may be specified together with `timer` to specify a void timer.

Otherwise, at most one type specifier may be given in a declaration; if omitted then it is taken to be `int`.

Types may also be qualified, to indicate special properties of the objects being declared.

type-qualifier ::= `const`
| `volatile`
| `in`
| `out`
| `buffered`
| `streaming`

`const` may appear with any type specifier. A `const` object may be initialised, but not thereafter assigned or input to. No object may be qualified `volatile`. A compiler is required to recognise this qualifier and issue an appropriate error message.

`in` and `out` may appear with the `port` and `port:`*n* type specifiers but not with `void`. An object qualified `in` may appear in input operations only, and an object qualified `out` may appear in output operations only (§A.8.3). `buffered` may appear with the `port` and `port:`*n* type specifiers. `streaming` may appear with the `chan` and `chanend` type specifiers.

Automatic variables may not be declared with type `port`, `port:`*n*, `chanend` or `core`. Static variables may not be declared with types `chan` or `chanend`. Ports specified with `void` may not be used in input or output operations.

A.7.3 `inline` specifier

Types may be specified `inline`, to suggest that calls to the function be as fast as possible. A definition is an *inline definition* if all the file-scope declarations for a function in the translation unit include the `inline` specifier without `extern`. An inline definition of a function with external linkage must not contain a definition of a modifiable object with static storage, and must not contain a reference to an identifier with external linkage.

The extent to which suggestions are effective is implementation-defined.

A.7.4 Structure and Union Declarations

A structure is an object consisting of a sequence of named members of various types. A union is an object that contains, at different times, any one of several members of various types. Structures and unions have the same form.

struct-or-union-specifier ::= *struct-or-union identifier*$_{opt}$ `{` ⟨*member*⟩$^+$ `}`
| *struct-or-union identifier*

struct-or-union ::= `struct`
| `union`

A *member* is a declaration for a member of the structure or union.

member ::= ⟨*specifier-or-qualifier*⟩$^+$ *struct-var-declarator-list* `;`

specifier-or-qualifier ::= *type-specifier*
| *type-qualifier*

struct-var-declarator-list ::= *var-declarator*
| *var-declarator* `,` *struct-var-declarator-list*

A type specifier of the form

struct-or-union identifier `{` ⟨*member*⟩$^+$ `}`

declares the identifier to be the *tag* of the structure or union specified by the member list. A subsequent declaration may refer to the same type by using the tag in a specifier without the member list:

struct-or-union identifier

If a specifier with a tag but without a list appears when the tag is not declared, an *incomplete type* is specified. Objects with an incomplete structure or union type may be used in contexts where their size is not needed. The type becomes complete on occurrence of a subsequent specifier with that tag, and containing a declaration list. Even in specifiers with a list, the structure or union type being declared is incomplete within the list, and becomes complete only at the `}` terminating the specifier.

A structure or union may not contain a member of incomplete or resource type, except that a structure may contain a member of type port or timer. If a structure is declared to have a member with one of these types then variables of the structure may be declared only as external declarations (see §A.9).

A structure or union specifier with a list but no tag creates a unique type; it can be referred to directly only in the declaration of which it is a part.

The names of members and tags do not conflict with each other or with ordinary variables. A member name may not appear twice in the same structure or union, but the same member name may be used in different structures or unions.

The members of a structure have addresses increasing in the order of their declarations. A member of a structure is aligned at an addressing boundary depending on its type.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most one of the members can be stored in a union at any time.

If a union contains several structures that share a common initial sequence, and if the union currently contains one of these structures, it is permitted to refer to the common initial part of any of the contained structures.

> Bit-fields are unsupported.

A.7.5 Enumerations

Enumerations are unique types with values ranging over a set of named constants called enumerators. The form of an enumeration specifier borrows from that of structures and unions.

enum-specifier ::= `enum` *identifier*$_{opt}$ `{` *enumerator-list* `}`
| `enum` *identifier*$_{opt}$ `{` *enumerator-list* `,` `}`
| `enum` *identifier*

enumerator-list ::= *enumerator*
| *enumerator-list* `,` *enumerator*

enumerator ::= *identifier*
| *identifier* = *constant-expression*

The enumerator type is compatible with `int`; identifiers in an enumerator list are declared as constants of type `int`, and may appear wherever constants are required. If no enumerators with = appear, then the values of the corresponding constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with = gives the associated identifier the value specified; subsequent identifiers continue the progression from the assigned value.

Enumerator names in the same scope must all be distinct from each other and from ordinary variable names, but the values need not be distinct.

The identifier in the *enum-specifier* names a particular enumeration. The rules for enum specifiers with and without tags and lists are the same as those for structure or union specifiers, except that incomplete enumeration types do not exist; the tag of an *enum-specifier* without an *enumerator-list* must refer to an in-scope specifier with a list.

A.7.6 Declarators

Declarators have the syntax:

var-declarator ::= *identifier* ⟨*dimension-size*⟩*
| `&` *identifier*
| `?` *indentifier* ⟨*dimension-size*⟩*
| `&` `?` *identifier*

fnc-declarator ::= *identifier* `(` *parameter-type-list*$_{opt}$ `)`

dimension-size ::= `[` *constant-expression*$_{opt}$ `]`

The structure of declarators resembles that of function and array expressions; the grouping is the same.

A.7.7 Meaning of Declarators

One or more declarators appear after a sequence of storage class and type specifiers. The declarators may be prefixed by either `select` or `transaction`, in which case only storage class specifiers are permitted as the declaration specifiers; the return type is implicitly `void`. Each declarator declares a unique main identifier. The storage class specifiers apply directly to this identifier, but its type depends on its form.

Considering only the type parts of the declaration specifiers (§ A.7.2), the optional `transaction` and `select`, and a particular declarator, a declaration has the form "*opt-transaction-or-select* `T D`" where `T` is a type and `D` is a declarator. The type attributed to the identifier in the various forms of declarator is described using this notation.

In a declaration `T D` where `D` is an unadorned identifier, the type of the identifier is `T`.

A port may be declared as an external declaration (see §A.9) or as a parameter only. A channel may be declared as a local variable only and a channel end may be declared as a parameter only. A structure containing a member or, recursively, any submember of resource type may be declared as an external declaration only.

A.7.7.1 Array Declarators

In a non-parameter declaration `T D` where `D` has the form

identifier `[`*constant-expression*`]`

and the type of the identifier in the declaration `T` *identifier* is "*type-modifier* `T`," the type of the identifier of `D` is "*type-modifier n*-array of `T`," where *n* is the result of evaluating the constant expression and specifies the number of elements in the array. The constant expression must have integral type, and value greater than 0.

In a parameter declaration `T D` where `D` has the form

identifier `[`*constant-expression*`]`

and the type of the identifier in the declaration `T` *identifier* is "*type-modifier* `T`," the type of the identifier of `D` is "reference to *type-modifier n*-array of `T`," where *n* is the result of evaluating the constant expression and specifies the number of elements in the array. The constant expression must have integral type, and value greater than 0.

In a declaration `T D` where `D` has the form

identifier `[]`

and the type of the identifier in the declaration `T` *identifier* is "*type-modifier* `T`," the type of the identifier of `D` is "*type-modifier incomplete*-array of `T`."

An array may be constructed from an arithmetic type, from a structure or union, from a port, timer, channel or channel end, or from another array (to generate a multi-dimensional array). Any type from which an array is constructed must be complete; it must not be an array or structure of incomplete type. This implies that for a multi-dimensional array, only the first dimension may be missing. The type of an object of incomplete array type is completed either by another, complete, declaration for the object (§A.9.2), or by initialising it (§A.7.8) or, for a function parameter in which the first dimension is missing, at run-time on entry to the function by the caller.

If `E1` is an array and `E2` an integer, then `E1[E2]` refers to the `E2`th member of `E1`. Arrays are stored by rows (last subscript varies faster) so that the first subscript in the declaration helps determine the amount of storage consumed by an array, but plays no other part in subscript calculations.

A.7.7.2 Reference Declarators

In a declaration `T D` where `D` has the form

`&` *identifier*

and the type of the identifier in the declaration `T` *identifier* is "*type-modifier* `T`," the type of the identifier of `D` is "reference to *type-modifier* `T`."

A reference declared with `&` may have an arithmetic, structure or union type, and may only be declared as a function parameter.

A.7.7.3 Nullable Declarators

In a declaration `T D` where `D` has the form

`?` *identifier*

and the type of the identifier in the declaration `T` *identifier* is "*type-modifier* `T`," the type of the identifier of `D` is "nullable *type-modifier* `T`."

A nullable object declared with `?` may have a resource type or a reference type, and may only be declared as a function parameter.

If an object contains a reference to `null`, it is invalid to reference the object except as the argument to a function taking a nullable parameter, as the operand of the `isnull` operator, or as the operand of the `sizeof` operator.

A.7.7.4 Function Declarators

In a function declaration `T D` where `D` has the form

`D1`*(parameter-type-list)*

and the type of the identifier in the declaration `T D1` is "*type-modifier* `T`," the type of the identifier of `D` is "*type modifier* function with arguments *parameter-type-list* returning `T`." If `T` has the form $\{T_1,\ldots,T_n\}$ then the return type is modified to read "list of types $T_1,\ldots,T_n$." In a function declaration `transaction void D` where `D` has the form

`D1`*(parameter-type-list)*

and the type of the identifier in the declaration `T D1` is "*type-modifier* `T`," the type of the

identifier of `D` is "*type modifier* transaction function with arguments *parameter-type-list* returning `void`." In a function declaration `select void D` where `D` has the form

`D1`*(parameter-type-list)*

and the type of the identifier in the declaration `T D1` is "*type-modifier* `T`," the type of the identifier of `D` is "*type-modifier* select function with arguments *parameter-type-list* returning `void`."

The syntax of the parameters is:

parameter-type-list ::= *parameter-list*
| *parameter-list* , *parameter-declaration*

parameter-list ::= *parameter-declaration*
| *parameter-list* , *parameter-declaration*

parameter-declaration ::= ⟨*dec-specifier*⟩$^+$ *abstract-or-void-dec*

abstract-or-void-dec ::= *var-declarator*
| *abstract-var-declarator*

The parameter list specifies the types of the parameters. As a special case, the declarator for a function with no parameters has a parameter list consisting solely of the keyword `void`. This is also signified by an empty parameter list. If the parameter type list ends with an ellipsis "`, ...`", then the function may accept more than the number of parameters explicitly described; see §A.6.3.2.

The only storage class specifier permitted in a parameter's declaration specifier is `register`, and this specifier is ignored unless the function declarator heads a function definition. This specifier has no effect on the behaviour of the function; the extent to which suggestions made by using this specifier are effective is implementation-defined.

Similarly, if the declarators contain identifiers and the function declarator does not prefix a function definition, the identifiers go out of scope immediately. Abstract declarators, which do not mention the identifiers, are discussed in §A.7.9.

A function declared with the storage class specifier `service` may declare only variables of type `chanend`.

Old-style C function declarations (see K&R §A8.6.3) are unsupported.

A.7.8 Initialisation

When an object is declared, its *init-var-declarator* may specify an initial value for the identifier being declared. The initialiser is preceded by `=`, and is either an expression, or a list of initialisers nested in braces.

initialiser ::= *on-statement*$_{opt}$ *expression*
| `{` *initialiser-list* `}`
| `{` *initialiser-list* `,` `}`

initialiser-list ::= *initialiser*
| *initialiser-list*

On statements are discussed in §A.8.8. If more than one `on`-statement is used with the same variable declaration, then all of these statements must refer to the same core.

All the expressions in the initialiser for a static object or array must be constant expressions as described in §A.6.19. The expressions in the initialiser for an `auto` or `register` object must likewise be constant expressions if the initialiser is a brace-enclosed list. However, if the initialiser for an automatic object is a single expression, it need not be a constant expression, but must have appropriate type for assignment to the object.

Timers, channels and cores must not be explicitly initialised. Timers not declared `extern` are initialised, at run-time, to refer to an unaliased hardware timer. Channels not declared `extern` are initialised, at run-time, to refer to two unaliased hardware channel ends that are connected together to create a point-to-point communication link. Ports not declared `extern`, and not explicitly initialised, are initialised with an implementation-defined value.

A static object that is not a timer, channel or port, and is not explicitly initialised, is initialised as if its expression (or its members) were assigned the constant 0. The initial value of an automatic object with arithmetic type not explicitly initialised is undefined.

The initialiser for an object of arithmetic type is a single expression, possibly in braces. The expression is assigned to the object. The initialiser for a port is a single constant expression, possibly in braces. The expression is assigned to the object; its interpretation and validity is implementation-defined.

The initialiser for a structure is either an expression of the same type, or a brace-enclosed list of initialisers for its members in order. If there are fewer initialisers in the list than members of the structure, the trailing members are initialised with 0. There may not be more initialisers than members.

The initialiser for an array is a brace-enclosed list of initialisers for its members. If the array has unknown size, the number of initialisers determines the size of the array, and its type becomes complete. If the array has fixed size, the number of initialisers may not exceed the number of members of the array; if there are fewer, the trailing members are initialised with 0.

As a special case, a character array may be initialised by a string literal (braces are optional); successive characters of the string initialise successive members of the array. If the array has unknown size, the number of characters in the string, including the terminating null character, determines its size; if its size is fixed, the number of characters in the string, not counting the terminating null character, must not exceed the size of the array.

The initialiser for a union is either a single expression of the same type, or a brace-enclosed initialiser for the first member of the union.

An *aggregate* is a structure or array. If an aggregate contains members of aggregate type, the initialisation rules apply recursively. Braces may be elided in the initialisation as follows: if the initialiser for an aggregate's member that is itself an aggregate begins with a left brace, then the succeeding comma-separated list of initialisers initialises the members of the subaggregate; it is erroneous for there to be more initialisers than members. If, however, the initialiser for a subaggregate does not begin with a left brace, then only enough elements from the list are taken to account for the members of the subaggregate; any remaining members are left to initialise the next member of the aggregate of which the subaggregate is a part.

A.7.9 Type Names

In several contexts (to specify type conversions explicitly with a cast, in a reinterpretation, and to declare parameter types in function declarators) it is necessary to supply the name of

a data type. This is accomplished using a *type name*, which is syntactically a declaration for an object of that type omitting the name of the object.

type-name ::= ⟨*specifier-or-qualifier*⟩$^+$ *abstract-var-declarator*

abstract-var-declarator ::= ⟨*dimension-size*⟩*

A.7.10 Typedef

Declarations whose storage class specifier is `typedef` do not declare objects; instead they define identifiers that name types (called typedef names).

typedef-name ::= *identifier*

A `typedef` declaration attributes a type to each name among its declarators in the usual way (see §A.7.7). Thereafter, each such typedef name is syntactically equivalent to a type specifier keyword for the associated type. `typedef` does not introduce new types, only synonyms for types that could be specified in another way. Typedef names may be redeclared in an inner scope, but a non-empty set of type specifiers must be given.

A.7.11 Type Equivalence

Two type specifier lists are equivalent if they contain the same set of type specifiers, taking into account that some specifiers can be implied by others (for example, `long` alone implies `long int`, `register` in formals is ignored). Structures and unions with different tags are distinct, and a tagless structure or union specifies a unique type.

Two types are the same if their abstract declarators (§A.7.9), after deleting any function parameter identifiers, are the same up to equivalence of type specifier lists. Array sizes (including the size of array parameters) are significant. For each parameter qualified `const` that is not a reference type, its type for this comparison is the unqualified version of its type.

A.8 Statements

Except as described, statements are executed in sequence. Statements are executed for their effect, and do not have values. They fall into several groups.

statement ::= *simple-statement*$_{opt}$;
| *compound-statement*
| *selection-statement*
| *iteration-statement*
| *jump-statement*
| *parallel-statement*
| *transaction-statement*

simple-statement ::= *expression-statement*
| *multiple-assignment*
| *input*
| *output*

A semicolon by itself is called a null statement; it is often used to supply an empty body to an iteration statement.

A.8.1 Expression Statement

The syntax of an expression statement is:

expression-statement ::= *expression*

Most expression statements are assignments or function calls. An expression statement must not have resource type. All side effects from the expression are completed before the next statement is executed.

A.8.2 Multiple Assignment Statement

The syntax of a multiple assignment statement is:

multiple-assignment ::= `{` *return-list* `}` *arithmetic-operator function-call*

return-list ::= *optional-variable*
| *optional-variable* , *return-list*

optional-variable ::= *variable-reference*
| `void`

The function must have return type "list of types $\texttt{T}_1,\ldots,\texttt{T}_n$" where n is the same as the number of optional variables in the return list.

The rules for assignment (see §A.6.17) apply to each of the variables in the return list: the ith value returned by the function replaces that of the object referred to by the ith optional variable reference. If the optional variable reference is `void` then the value is discarded.

A variable which is changed in the subscript of an optional variable may not appear in any other optional variable or in the function call, including the arguments to the function. A variable which is changed in the function call, including arguments to function may not appear in any optional variable. These rules apply recursively to variables which are changed or appear in functions called in the optional variables or the function call.

A variable which is changed by the assignment may not appear in any other optional variable or recursively appear in functions called in any other optional variable.

If any of the objects assigned to are the same as one another then the assignment is invalid.

A.8.3 Input and Output Statements

An input statement receives a value from a channel end, port or timer, and assigns the received value to an object.

input ::= *resource* time_{opt} $\mathit{predicate}_{opt}$ *input-operator*
dest $\mathit{input\text{-}timestamp}_{opt}$

resource ::= *variable-reference*

time ::= `@` *expression*

input-operator ::= `:>`
| `:> >>`

dest ::= *declared-var-reference*

input-timestamp ::= `@` *declared-var-reference*

declared-var-reference ::= ⟨*declaration-specifier*⟩$^+$ *identifier*$_{opt}$
| *variable-reference*

predicate ::= `when` *function-call*

The resource must name either a channel end, port or timer. If the resource names a channel end or timer then neither a *time* nor an *input-timestamp* is allowed. If the resource names a channel end then a *predicate* is not allowed. If the resource names a port then the port must not be qualified `out` and the destination variable must have arithmetic type.

If a *time* is provided then the time expression must have arithmetic type. The input is said to be *timed*.

If an *input-timestamp* is provided then the *declared-var-reference* must name a modifiable lvalue with arithmetic type. The input is said to be *timestamped*.

If a *predicate* is provided then the named function must have been declared to return `void` and from its parameter list there must be precisely one port or timer declaration, which must be qualified `void`. The input is said to be *predicated*. The supported predicates are implementation-defined. The function call is shortcutted: the resource variable must not be passed as an argument; it is passed implicitly as the port or timer argument.

A *declared-var-reference* must be a modifiable lvalue if an identifier is named. It must not define a new type. If the resource names a port or timer then the lvalue must not reference a structure or union; if no identifier is given then the type must not specify a structure or union, but it may specify `void`. If no declaration specifiers are provided then the type of the variable must not be qualified with `const`; if it is a structure or union, it must not have any member or, recursively, submember qualified with `const`. If any declaration specifiers are provided then the variable reference is also a declaration; the specifiers must not contain `typedef` but may contain `const`.

A variable which is changed by any part of the input may not, except as described below, appear in any other part of the input. If the *declared-var-reference* is a *variable-reference* then the identifier named may appear in any other parts of the input, as long as the variables named by the identifiers in these parts are not changed. Additionally, the variable which is written by the *input-timestamp* may not appear in the *dest*, and the variable which is written by the *dest* may not appear in the *input-timstamp*. These rules apply recursively to all variables which are changed in functions called by the input.

The first variable declared in an input begins an inner scope which is understood to begin immediately preceding the declaration and which persists to the end of the input. If the input appears in the case of a select then this scope continues to persist to the end of the statement list after the colon.

If the resource names a channel end or timer, or the destination identifier is omitted, then the `:> >>` operator is not allowed.

An output statement transmits the value of an expression to a channel end or port.

output ::= *resource time*$_{opt}$ *output-operator*
expression output-timestamp$_{opt}$

resource ::= *variable-reference*

time ::= `@` *expression*

output-operator ::= `<:`
| `<: >>`

output-timestamp ::= `@` *variable-reference*

The resource must name a channel end or port. If the resource names a channel end then neither a *time* nor a *output-timestamp* is allowed. If the resource names a port then the port must not be qualified `in` and the output expression must have arithmetic type; otherwise the output expression must either have arithmetic type, or must be a structure or union.

If the resource names a channel end then the `<: >>` operator is not allowed. If the `<: >>` is specified then the output expression must be a modifiable lvalue.

If a *time* is provided then an *output-timestamp* is not allowed. The time expression must have arithmetic type. The output is said to be *timed*.

If an *output-timestamp* is provided then the variable reference must be a modifiable lvalue with arithmetic type. The output is said to be *timestamped*.

A variable which is changed by any part of the output may not, except as described below, appear in any other part of the output. The identifier named by the *output-timestamp* may appear in any other parts of the output as long as the variables named by the identifiers in these parts are not changed. These rules apply recursively to all variables which are changed in functions called by the output.

> Input and output statements are new; I/O operations are conventially performed using interrupts and system calls (via library routines in C).

A.8.3.1 Channel Input and Output

An input on a channel end causes the processor to wait until a matching outputter is ready in a parallel statement (see §A.8.8) before receiving a value. If the type of an input variable is specified but the identifier is missing then the received value is ignored. See §A.11 for the meaning of an input in a channel communication.

An output on a channel causes the processor to wait until a matching inputter is ready in a parallel statement before sending the value. See §A.11 for the meaning of an output in a channel communication.

A.8.3.2 Port Input and Output

An input from a port causes the specified port to provide the processor a value. If the port transfer width is *w* bits, these *w* bits are assigned to the least significant bits of a variable with the port's notional transfer type (see §A.3.2) with any remaining bits being set to zero. If the type of an input variable is specified but an identifier is missing, or if a `void` type is specified, then this input variable is ignored. If the input is used with the `:> >>` operator, the destination variable is right-shifted by *w* bits and the bitwise inclusive-or of this value and the input variable is then assigned to the destination variable; otherwise the input variable is assigned to the destination variable.

If a `when` condition is provided, the function and its arguments are provided to the port before performing the input.

An output to a port causes the output expression to be first cast to the port's notional transfer type and then provided to the port. If the output is used with the `<: >>` operator, the output variable is then right shifted by *w* bits.

If the input or output is timed, the value specified by *time* is cast to the port's notional counter type prior and provided to the port before performing the input or output.

If the input or output is timestamped, *t* bits are assigned to the least significant bits of a variable with the port's notional counter type (see §A.3.2) with any remaining bits being set to zero; this variable is then assigned to the timestamp variable.

See Appendix B for the meaning of inputs and outputs on ports with respect the communication performed between the port and procecssor, and the corresponding operation performed by the port on its pins.

A.8.3.3 Timer Input

An input from a timer causes the timer to provide the current value of its counter. This value is assigned to the least significant bits of a variable with the timer's notional counter type (see §A.3.2) with any remaining bits being set to zero. If the type of an input variable is specified but an identifier is missing, or if a `void` type is specified, then this input variable is ignored; otherwise the input variable is assigned to the destination variable.

A.8.4 Compound Statement

So that several statements can be used where one is expected, the compound statement (or "block") is provided. The body of a function definition is a compound statement.

compound-statement ::= `{` ⟨*var-declaration*⟩* ⟨*statement*⟩* `}`

If an identifier in the *var-declaration-list* was in scope outside the block, the outer declaration is suspended within the block (see §A.10.1). An identifier may be declared only once in the same block. These rules apply to identifiers in the same name space (§A.10); identifiers in different name spaces are treated as distinct.

Initialisation of automatic objects is performed each time the block is entered at the top, and proceeds in the order of the declarators. Initialisation of `static` objects is performed only once, before the program begins execution.

A.8.5 Selection Statements

Selection statements choose one of several flows of control.

selection-statement ::= `if (` *expression* `)` *statement*
| `if (` *expression* `)` *statement* `else` *statement*
| `switch (` *expression* `) {` ⟨*labelled-statement*⟩$^+$ `}`
| `select {` ⟨*guarded-statement*⟩$^+$ `}`

labelled-statement ::= `case` *constant-expression* `:` ⟨*statement*⟩*
| `default :` ⟨*statement*⟩*

guarded-statement ::= `case` $replicator_{opt}$ $enable\text{-}exp_{opt}$ *input* `:` ⟨*statement*⟩*
| `case` $replicator_{opt}$ $enable\text{-}exp_{opt}$ *function-call* `:`
⟨*statement*⟩*
| `case` $replicator_{opt}$ $enable\text{-}exp_{opt}$ *slave-statement* `:`
⟨*statement*⟩*
| `default :` ⟨*statement*⟩*
| `case` *function-call* `;`

replicator ::= `( int` *variable* `=` *expression* `;` *expression* `;` *experssion* `)`

enable-exp ::= *expression* `=>`

In both forms of the `if` statement, the expression, which must have arithmetic type, is evaluated, including all side effects, and if it compares unequal to 0, the first substatement is executed. In the second form, the second substatement is executed if the expression is 0. The `else` ambiguity is resolved by connecting an `else` with the last encountered `else`-less `if` at the same block nesting level.

The `switch` statement causes control to be transferred to one of several case statements depending on the value of the expression, which must have integral type. The controlling expression undergoes integral promotion (§A.5.1), and the case constants are converted to the promoted type. No two of the case constants in the same switch may have the same value after conversion. There may also be at most one `default` label associated with a switch.

When the `switch` statement is executed, its expression is evaluated, including all side effects, and compared with each case constant. If one of these case constants is equal to the value of the expression, control passes to the statement of the matched `case` label. If no case constant matches the expression, and if there is a `default` label, control passes to the default-labelled statement. If no case matches, and if there is no `default`, then none of the substatements of the switch is executed.

The `select` statement causes control to be transferred to one of several guarded case statements. A guarded statement may consist of an optional replicator and an optional expression followed by an input (§A.8.3), a slave transaction statement (§A.8.9) or a function call, followed by a colon and a list of zero or more statements.

In a replicator, the third expression must either add or subtract a constant expression to the variable declared by the replicator. A replicator is short-hand for multiple cases, and has the same meaning as if the code was expanded as with a `for` loop. In addition, if the initialiser is a constant expression and the second expression is a relational expression that compares the variable declared by the replicator to a constant expression, the variable declared by the replicator is treated as a constant expression in the replicator body. The declared variable may not be modified outside of the replicator.

If the statement before the colon is a call to a transaction function (§A.9.1.1) then this is considered shorthand for a slave transaction statement that performs the call. The enable expression must have arithmetic type, and it must not modify a local variable, static variable or reference parameter; any functions called within the expression, recursively, must not modify a static variable, reference parameter, or perform an input or output. The modification rules that apply to the enable expression also apply to the arguments of a call to a select function; the rules also apply to an input statement that appears before the colon, except that the input lvalue is (by definition) modified. An input guard that causes any observable behaviour on a port prior to being selected is invalid. There may be at most one `default` label associated with a select.

A guarded statement may also consist of a call to a select function (see §A.9.1.2) followed by a semicolon. The rules that apply to the enable expression also apply to the arguments of a call to a select function. The ports, timers and channel ends named before each colon, and as arguments to a select function, must be distinct.

When the `select` statement is executed, each guard that contains no enable expression is enabled. For each guard containing an enable expression, the expression is evaluated and, if it compares unequal to 0, the case is enabled. The behaviour of a call to a select function is the same as if the cases of the select function were included inline in the select.

Following the enabling sequence, if no cases are enabled then either the `default` case is executed, if provided, or none of the substatements of the select is executed and the select

never completes (it deadlocks). Otherwise, the select waits until an input or transaction in one of the enabled cases is ready and performs the corresponding input or transaction. If more than one of these inputs or transactions is ready then the choice of which is executed is made nondeterministically.

After performing an input or transaction, the statements following the colon of the selected case are executed.

The statements after the colon in each `select` case statement must terminate with a `break` or `return`, so that control never flows from one case statement to the next.

A.8.6 Iteration Statements

Iteration statements specify looping.

iteration-statement ::= `while (` *expression* `)` *statement*
| `do` *statement* `while (` *expression* `) ;`
| `for (` *for-init*$_{opt}$ `;` *expression*$_{opt}$ `;` *simple-list*$_{opt}$ `)`
statement

for-init ::= *var-declaration*
| *simple-list*

simple-list ::= *simple-statement*
| *simple-statement* `,` *simple-list*

In the `while` and `do` statements, the substatement is executed repeatedly so long as the value of the expression remains unequal to 0; the expression must have arithmetic type. With `while`, the test, including all side effects from the expression, occurs before each execution of the statement; with `do`, the test follows each iteration.

A `for` statement may declare a variable (see A.3), whose scope begins immediately after the declaration and persists to the end of the statement; if present, the variable initialiser is evaluated once. Alternatively, if a list of simple statements is provided, the statements are executed once. The expression must have arithmetic type; it is evaluated before each iteration, and if it is equal to 0, the `for` is terminated. The optional list of simple statements following the second semicolon is evaluated after each iteration. Any of these three components may be dropped; a missing test expression makes the implied test equivalent to testing a non-zero constant.

A.8.7 Jump Statements

Jump statements transfer control unconditionally.

jump-statement ::= `continue ;`
| `break ;`
| `return` *expression*$_{opt}$ `;`
| `return {` *expression-list* `} ;`

A `continue` statement may appear only within an iteration statement, and may not appear in a parallel, master or slave statement, unless that statement contains an iteration statement in which it is enclosed. It causes control to pass to the loop-continuation portion of the smallest enclosing such statement.

A `break` statement may appear only in an iteration statement, a `switch` statement or a `select` statement, and may not appear in a parallel, master or slave statement, unless that statement contains an iteration, `switch` or `select` statement in which it is enclosed. It terminates execution of the smallest enclosing such statement; control passes to the statement following the terminated statement.

A function returns to its caller by the `return` statement. A `return` statement may not appear in a parallel, master or slave statement. When `return` is followed by an expression, the value is returned to the caller of the function. The expression is converted, as if by assignment, to the type returned by the function in which it appears.

When `return` is followed by an list of expressions in braces, the list of values is returned to the caller of the function. For a return with *n* expressions, the return type of the function must be "list of types $\texttt{T}_1,\ldots,\texttt{T}_n$." For all expressions ($i$=1..$n$), the ith expression is converted, as if by assignment, to the ith type returned by the function in which it appears.

Flowing off the end of a function is equivalent to a return with no expression. In either case, the returned value is undefined.

> `goto` is unsupported.

A.8.8 Concurrency Statement

So that several statements can be executed concurrently, the parallel statement is provided.

parallel-statement ::= `par` $replicator_{opt}$ `{` ⟨*thread*⟩* `}`

replicator ::= `(` `int` *variable* = *expression* `;` *expression* `;` *experssion* `)`

thread ::= $on\text{-}statement_{opt}$ *statement*

on-statement ::= `on` *variable-reference* `:`

Replicators are discussed in §A.8.5. In addition, the initialiser must assign a constant expression, and the second expression must be a relational expression that compares the variable declared by the replicator to a constant expression. The relation operator may not be equality or inequality and the condition must be satisfiable for some value of the declared variable (for example, `x > MAX_INT` is disallowed).

An on-statement is only permitted if it appears in a parallel-statement that is either the only statement in the enclosing function, or if it is one of two statements of a function compound-statement, the second being a return statement that returns a constant expression that evaluates to 0.

`on` does not change the behaviour of the statement it prefixes.

Values may be passed between concurrent statements by communication on channels (§A.3.2) using input and output statements (§A.8.3).

Variables and channels used in parallel statements are subject to usage rules which prevent them from being accidentally shared between statements in potentially dangerous ways, as described below.

A variable which is changed by assignment or input in one of the statements of a `par` may not appear in any other statement of the `par`. This rule applies recursively to all variables which are changed by assignment or input in a function that is called by a statement of a `par`. (By implication, a variable may appear in expressions in any number of statements of a `par` so long as it is not assigned or input in any of these statements.)

A channel may not be used in more than two statements of a `par`. Channel ends, ports and timers may not be used in more than one statement of a `par`.

If a statement contains of a number of sub-statements, such as a compound-statement (§A.8.4), then all of the sub-statements are considered together as a single statement for the purpose of this rule.

A.8.9 Transaction Statement

So that several communications over a channel can be logically grouped together, the transaction statement is provided.

transaction-statement ::= *slave-statement*
| *master-statement*

slave-statement ::= `slave` *statement*

master-statement ::= `master` *statement*

All inputs and outputs within `master` or `slave` are logically part of the same transaction; the extent to which the underlying communication protocols are optimised for transaction communications is implementation-defined.

The statements must reference precisely one channel end, which is said to be the *transactor*. If the variable reference designating the transactor contains any array indices then the indices must be constant expressions. The transactor must not name a streaming channel.

Within a transaction statement, inputs and outputs on any channel end other than the transactor is prohibited; using a channel end other than the transactor as an argument to a function is prohibited; using the transactor as an argument to a function that is not a transaction function is prohibited; introducing a nested transaction statement is prohibited; and declaring a channel (in the statement or, recursively, in any function called within the transaction) is prohibited.

A.9 External Declarations

The unit of input provided to the XC compiler is called a translation unit; it consists of a sequence of external declarations, which are either declarations or function definitions.

translation-unit ::= ⟨*external-declaration*⟩$^+$
external-declaration ::= *declaration*
| *function-definition*

The scope of external declarations persists to the end of the translation unit in which they are declared.

A.9.1 Function Definitions

Function definitions have the form:

function-definition ::= *fnc-declaration compound-statement*
| *trn-declaration compound-statement*
| *sel-declaration* `{` ⟨*guarded-statement*⟩$^+$ `}`

The only storage-class specifiers allowed among the declaration specifiers are `extern`, `static` or `inline`; see §A.10.2 for the effect. The ellipses "`, ...`" operator is not allowed in function definitions.

A function may return an arithmetic type, a structure, a union or `void`, but not a resource type, a function or an array. Alternatively it may return a list of any combination of arithmetic types, structures and unions. A function may not return a structure containing a member or, recursively, any submember of resource type.

Unless the parameters consist solely of `void`, indicating that the function takes no parameters, each declarator in the parameter list must contain an identifier. The parameters are understood to be declared just after the beginning of the compound statement constituting the function's body, and thus the same identifiers must not be redeclared there (although they may be redeclared in inner blocks). During the call to a function, the arguments are converted as necessary and assigned to the parameters; see §A.6.3.2.

A.9.1.1 Transaction Functions

A function declaration modified by the keyword `transaction` is a transaction function (see §A.7.7.4). The function body consists of a list of statements, which is by definition a transaction statement (see §A.8.9). The function must declare precisely one channel end in its parameter list, which is by definition the transactor.

A.9.1.2 Select Functions

A function declaration modified by the keyword `select` is a select function (see §A.7.7.4). The function body consists of a list of guarded statements, which is by definition a select statement (see §A.8.5). The guards of a select function may not contain replicators or transactors.

A.9.2 External Declarations

External declarations specify the characteristics of objects, functions and other identifiers. The term "external" refers to their location outside functions, and is not directly connected with the `extern` keyword; the storage class for an externally-declared object may be left empty, or it may be specified as `extern` or `static`.

Several external declarations for the same identifier may exist within the same translation unit if they agree in type and linkage, and if there is at most one definition for the identifier.

Two declarations for an object or function are deemed to agree in type under the rules discussed in §A.7.11. In addition, if the declarations differ because one type is an incomplete structure or union and the other is the corresponding completed type with the same tag, the types are taken to agree. If one type is an incomplete array type (§A.7.7.1) and the other is a completed array type, the types, if otherwise identical, are also taken to agree.

If the first external declaration for a function or object includes the `static` specifier, the identifier has *file-scope (internal linkage)*; otherwise it has *program-scope (external linkage)*. Linkage is discussed in §A.10.2.

An external declaration for an object is a definition if it has an initialiser. An external object declaration that does not have an initialiser, and does not contain the `extern` specifier, is a *tentative definition*. If a definition for an object appears in a translation unit, any tentative

definitions are treated as redundant declarations. If no definition for the object appears in the translation unit, all its tentative definitions become a single definition with initialiser 0.

Each object must have exactly one definition. For objects with internal linkage, the rules apply separately to each translation unit. For objects with external linkage, it applies to the entire program.

A.10 Scope and Linkage

There are two kinds of scope to consider: first, the *lexical scope* of an identifier, which is the region of the program text within which the identifier's characteristics are understood; and second, the scope associated with objects with external linkage, which determines the connections between identifiers in separately compiled translation units.

A.10.1 Lexical Scope

Identifiers fall into several name spaces that do not interfere with one another; the same identifier may be used for different purposes, even in the same scope, if the uses are in different name spaces. These classes are: objects and functions; tags of structures and unions; and members of each structure or union individually.

The lexical scope of an object or function identifier in an external declaration begins at the end of its declarator and persists to the end of the translation unit in which it appears. The scope of a parameter of a function definition begins at the start of the block defining the function, and persists through the function; the scope of a parameter in a function declaration ends at the end of the declarator. The scope of an identifier declared at the head of a block begins at the end of its declarator, and persists to the end of the block. The scope of a structure or union begins at its appearance in a type specifier, and persists to the end of the translation unit (for declarations at the external level) or to the end of the block (for declarations within a function).

If an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of the identifier outside the block is suspended until the end of the block.

A.10.2 Linkage

Within a translation unit, all declarations of the same object or function identifier with internal linkage refer to the same thing, and the object or function is unique to that translation unit. All declarations for the same object or function identifier with external linkage refer to the same thing, and the object or function is shared by the entire program.

The first external declaration for an identifier gives the identifier internal linkage if the `static` specifier is used, external linkage otherwise.

An inline definition (§A.7.3) does not provide an external definition for the function and does not forbid an external definition. An inline definition provides an alternative to an external definition which may be used it implement any call to the function in the same translation unit. It is unspecified whether a call to the function uses the inline definition or the external definition.

A.11 Channel Communication

A channel communication occurs when, on the same channel,

- an output is executed in parallel with an input, or
- a master transaction is executed in parallel with a slave transaction.

An output executed in parallel with a slave transaction is invalid; a master transaction executed in parallel with an input is invalid.

Outside a transaction, an output-input communication in which the number of bytes output is unequal to the number of bytes input is invalid. Inside a transaction, if all communications are valid individually then the transaction is also valid. Additionally, if a communication occurs in which the number of bytes output is unequal to the number of bytes input then whether or not the transaction is invalid, and the value communicated is implementation-defined.

An invalid communication within a transaction need not cause the transaction to become invalid until slave transaction statement goes out of scope.

The meaning of an output-input communication in which the type of the output expression e is the same as the type of the input variable v is the same as the assignment $v = e$. If the types are different and the communication is not invalid then the meaning is the assignment $v = (e, type(v))$ (see §A.6.3.4).

A.12 Invalid Operations

An operation that is syntactically legal but for some reason or under some circumstances is semantically invalid may be treated in one of three ways:

- It may be reported as a compiler error.
- It may have implementation-defined behaviour, for example the processor could issue a trap, and a trap handler could terminate the program.
- It may result in undefined behaviour.

If at time t a program is guaranteed to execute some sequence of events that cause it to become invalid at some time in the future $t+n$ then it is permitted to become invalid any time during $[t..t+n]$. This allows an implementation to improve code efficiency, for example by relocating safety checks outside of loops.

A.13 Preprocessing

The preprocessor specification is defined to be the same as with C99 [7, §6.10], with the following exceptions:

- The macro `__XC__` is defined as 1.
- The macros `__STDC__`, `__STDC_HOSTED__` and `__STD_VERSION__` are not defined.

A.14 Grammar

Below is a summary of the grammar given throughout this appendix. The grammar has undefined terminal symbols *integer-constant*, *character-constant*, *identifier*, *string* and *enumeration-constant*; words and symbols wrtten in `typewriter` are terminals given literally.

translation-unit ::= ⟨*external-declaration*⟩$^+$

external-declaration ::= *declaration*
| *function-definition*

function-definition ::= *fnc-declaration compound-statement*
| *trn-declaration compound-statement*
| *sel-declaration* `{` ⟨*guarded-statement*⟩$^+$ `}`

declaration ::= *on-statement*$_{opt}$ *actual-declaration*

actual-declaration ::= *var-declaration*
| *fnc-declaration* ;
| *trn-declaration* ;
| *sel-declaration* ;

var-declaration ::= ⟨*dec-specifier*⟩* *init-var-declarator-list*$_{opt}$;

fnc-declaration ::= ⟨*dec-specifier*⟩* *fnc-declarator*
| `{` *dec-specifier-list* `}` *fnc-declarator*

trn-declaration ::= ⟨*dec-specifier*⟩* `transaction` *fnc-declarator*

sel-declaration ::= ⟨*dec-specifier*⟩* `select` *fnc-declarator*

dec-specifier-list ::= ⟨*dec-specifier*⟩*
| *dec-specifier-list* , ⟨*dec-specifier*⟩*

dec-specifier ::= *storage-class-specifier*
| *type-specifier*
| *type-qualifier*
| `inline`

storage-class-specifier ::= `auto`
| `register`
| `static`
| `extern`
| `typedef`
| `service`

type-specifier ::= `void`
| `char`
| `short`
| `int`
| `long`
| `signed`
| `unsigned`
| `chan`
| `chanend`
| `port`
| `port:`*n*
| `timer`
| `core`
| *struct-or-union-specifier*
| *enum-specifier*
| *typedef-name*

type-qualifier ::= `const`
| `volatile`
| `in`
| `out`
| `buffered`
| `streaming`

struct-or-union-specifier ::= *struct-or-union identifier*$_{opt}$ `{` ⟨*member*⟩$^{+}$ `}`
| *struct-or-union identifier*

struct-or-union ::= `struct`
| `union`

init-var-declarator-list ::= *init-var-declarator*
| *init-var-declarator-list* `,` *init-var-declarator*

init-var-declarator ::= *var-declarator* ⟨`=` *initialiser*⟩$_{opt}$

member ::= ⟨*specifier-or-qualifier*⟩$^{+}$ *struct-var-declarator-list* `;`

specifier-or-qualifier ::= *type-specifier*
| *type-qualifier*

struct-var-declarator-list ::= *struct-var-declarator*
| *struct-var-declarator-list* `,` *struct-var-declarator*

struct-var-declarator ::= *var-declarator*
| *var-declarator*$_{opt}$ `:` *constant-expression*

enum-specifier ::= `enum` *identifier*$_{opt}$ `{` *enumerator-list* `}`
| `enum` *identifier*$_{opt}$ `{` *enumerator-list* `,` `}`
| `enum` *identifier*

enumerator-list ::= *enumerator*
| *enumerator-list* , *enumerator*

enumerator ::= *identifier*
| *identifier* = *constant-expression*

var-declarator ::= *identifier* ⟨*dimension-size*⟩*
| *& identifier*
| ? *indentifier* ⟨*dimension-size*⟩*
| & ? *identifier*

fnc-declarator ::= *identifier* (*parameter-type-list*$_{opt}$)

dimension-size ::= [*constant-expression*$_{opt}$]

parameter-type-list ::= *parameter-list*
| *parameter-list* , *parameter-declaration*

parameter-list ::= *parameter-declaration*
| *parameter-list* , *parameter-declaration*

parameter-declaration ::= ⟨*dec-specifier*⟩$^{+}$ *abstract-or-void-dec*

abstract-or-void-dec ::= *var-declarator*
| *abstract-var-declarator*

initialiser ::= *on-statement*$_{opt}$ *expression*
| { *initialiser-list* }
| { *initialiser-list* , }

initialiser-list ::= *initialiser*
| *initialiser-list* , *initialiser*

type-name ::= ⟨*specifier-or-qualifier*⟩$^{+}$ *abstract-var-declarator*

abstract-var-declarator ::= ⟨*dimension-size*⟩*

typedef-name ::= *identifier*

statement ::= *simple-statement*$_{opt}$;
| *compound-statement*
| *selection-statement*
| *iteration-statement*
| *jump-statement*
| *parallel-statement*
| *transaction-statement*

simple-statement ::= *expression-statement*
| *multiple-assignment*
| *input*
| *output*

compound-statement ::= `{` ⟨*var-declaration*⟩* ⟨*statement*⟩* `}`

selection-statement ::= `if (` *expression* `)` *statement*
| `if (` *expression* `)` *statement* `else` *statement*
| `switch (` *expression* `) {` ⟨*labelled-statement*⟩+ `}`
| `select {` ⟨*guarded-statement*⟩+ `}`

labelled-statement ::= `case` *constant-expression* `:` ⟨*statement*⟩*
| `default :` ⟨*statement*⟩*

guarded-statement ::= `case` $replicator_{opt}$ $enable\text{-}exp_{opt}$ *input* `:` ⟨*statement*⟩*
| `case` $replicator_{opt}$ $enable\text{-}exp_{opt}$ *function-call* `:` ⟨*statement*⟩*
| `case` $replicator_{opt}$ $enable\text{-}exp_{opt}$ *slave-statement* `:` ⟨*statement*⟩*
| `default :` ⟨*statement*⟩*
| `case` *function-call* `;`

replicator ::= `(` `int` *variable* `=` *expression* `;` *expression* `;` *experssion* `)`

enable-exp ::= *expression* `=>`

iteration-statement ::= `while (` *expression* `)` *statement*
| `do` *statement* `while (` *expression* `) ;`
| `for (` $for\text{-}init_{opt}$ `;` $expression_{opt}$ `;` $simple\text{-}list_{opt}$ `)` *statement*

jump-statement ::= `continue ;`
| `break ;`
| `return` $expression_{opt}$ `;`
| `return {` *expression-list* `} ;`

parallel-statement ::= `par` $replicator_{opt}$ `{` ⟨*thread*⟩* `}`

thread ::= $on\text{-}statement_{opt}$ *statement*

on-statement ::= `on` *variable-reference* `:`

transaction-statement ::= *slave-statement*
| *master-statement*

slave-statement ::= `slave` *statement*

master-statement ::= `master` *statement*

for-init ::= *var-declaration*
| *simple-list*

simple-list ::= *simple-statement*
| *simple-list* `,` *simple-statement*

expression-statement ::= *expression*

expression ::= *assignment-expression*

assignment-expression ::= *conditional-expression*
| *variable-reference assignment-operator*
assignment-expression

assignment-operator ::= *one of*
= *= /= %= += -= <<= >>= &= ^= |=

conditional-expression ::= *logical-OR-expression*
| *logical-OR-expression* ?
expression : *conditional-expression*

constant-expression ::= *conditional-expression*

logical-OR-expression ::= *logical-AND-expression*
| *logical-OR-expression* || *logical-AND-expression*

logical-AND-expression ::= *inclusive-OR-expression*
| *logical-AND-expression* && *inclusive-OR-expression*

inclusive-OR-expression ::= *exclusive-OR-expression*
| *inclusive-OR-expression* | *exclusive-OR-expression*

exclusive-OR-expression ::= *AND-expression*
| *exclusive-OR-expression* ^ *AND-expression*

AND-expression ::= *equality-expression*
| *AND-expression* & *equality-expression*

equality-expression ::= *relational-expression*
| *equality-expression* == *relational-expression*
| *equality-expression* != *relational-expression*

relational-expression ::= *shift-expression*
| *relational-expression* < *shift-expression*
| *relational-expression* > *shift-expression*
| *relational-expression* <= *shift-expression*
| *relational-expression* >= *shift-expression*

shift-expression ::= *additive-expression*
| *shift-expression* << *additive-expression*
| *shift-expression* >> *additive-expression*

additive-expression ::= *multiplicative-expression*
| *additive-expression* + *multiplicative-expression*
| *additive-expression* - *multiplicative-expression*

multiplicative-expression ::= *cast-expression*
| *multiplicative-expression* * *cast-expression*
| *multiplicative-expression* / *cast-expression*
| *multiplicative-expression* % *cast-expression*

cast-expression ::= *unary-expression*
| `(` *type-name* `)` *cast-expression*

unary-expression ::= *postfix-expression*
| `++` *variable-reference*
| `--` *variable-reference*
| *unary-operator cast-expression*
| `sizeof` *unary-expression*
| `sizeof (` *type-name* `)`
| `isnull (` *unary-expression* `)`

unary-operator ::= *one of*
`+ - ~ !`

postfix-expression ::= *primary-expression*
| *variable-reference* `++`
| *variable-reference* `--`

primary-expression ::= *variable-reference*
| *function-call*
| *constant*
| *string*
| `(` *expression* `)`

multiple-assignment ::= `{` *return-list* `}` *assignment-operator function-call*

return-list ::= *optional-variable*
| *return-list* `,` *optional-variable*

optional-variable ::= *variable-reference*
| `void`

input ::= *resource* $time_{opt}$ $predicate_{opt}$ *input-operator*
dest $timestamp_{opt}$

resource ::= *variable-reference*

time ::= `@` *expression*

input-operator ::= `:>`
| `:> >>`

dest ::= *declared-var-reference*

input-timestamp ::= `@` *declared-var-reference*

output-timestamp ::= `@` *variable-reference*

declared-var-reference ::= $\langle$*declaration-specifier*$\rangle^{+}$ $identifier_{opt}$
| *variable-reference*

predicate ::= `when` *function-call*

output ::= *resource* $time_{opt}$ *output-operator*
expression $timestamp_{opt}$

output-operator ::= `<:`
| `<: >>`

function-call ::= *identifier* `(` $expression\text{-}list_{opt}$ `)`

variable-reference ::= *identifier*
| *variable-reference* `[` *expression* `]`
| *variable-reference* `.` *identifier*
| `(` *variable-reference* `,` *type-name* `)`

expression-list ::= *expression*
| *expression-list* `,` *expression*

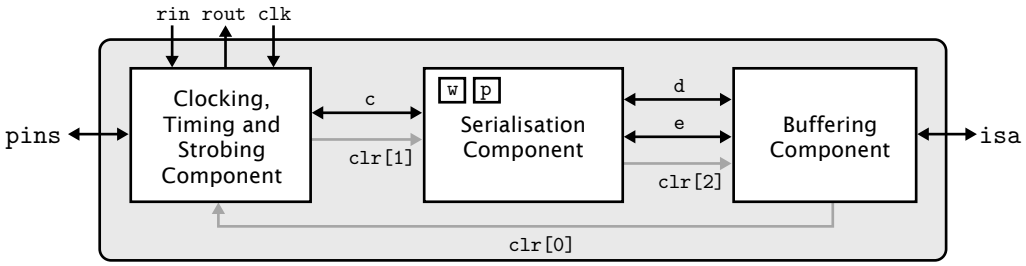
constant ::= *integer-constant*
| *character-constant*
| *enumeration-constant*
| `null`

APPENDIX B

XC I/O Specification

The specification given in this appendix describes the functional behaviour of I/O operations on ports.

For the purpose of describing semantics, a port can be defined as a collection of concurrent threads that perform a number of functions on data as it flows between the processor and the pins. The compositional model of an XMOS port is shown in the diagram below.



Logically, a port comprises three components that extend a “raw port” with functionality such as clocking, timing, strobing, serialisation and buffering. The XC program below defines a model for these components and their connectivity.

```
port pins, clk, rin, rout;
int w, p;
int direction;

void main(void) {
  chan c, d, e, isa, clr[3];
  par {
    clkTimeStrobe(pins, rin, rout, clk, c, clr[0], clr[1]);
    serialiser(w, p, c, d, e, clr[1], clr[2]);
    buffer(n, direction, d, e, isa, !isnull(rout), clr[2], clr[0]);
    processor(isa);
} }
```

By defining the functionality of ports in this way, the task of programming ports can be considered analogous to interfacing with other components in a concurrent system.

B.1 The Functional Model of Clocked I/O

The behaviour of inputting and outputting on a clocked, unbuffered port that uses no ready-in or ready-out signals is defined as follows:

- An output edge occurs on the next falling edge of the port's clock.
- An output causes data to be driven on the next output edge; the processor blocks until the subsequent rising edge.
- An input edge occurs on the next rising edge of the port's clock.
- An input causes data to be sampled by the port on the next input edge; the processor blocks until this time.

The function below implements these semantics. It operates on two raw ports that interface with clock and data signals. The clock is modelled as a pin that when toggled, for example by an XCore clock block, signifies a clock edge. (Note that the function does not implement any timing or strobing operations, and that it interfaces directly with the processor.)

```
void clkTimeStrobe(port pins, port clk, chanend isa) {
  pwidth_t data   = 0;
  int      clkVal = 0;
  int      state  = QUIET;

  while (1) {
    select {
      case clk when pinsneq(clkVal) :> clkVal :
        if (!clkVal && state == OUTPUT) {
          pins <: data;
          state = PENDQUIET;
        }
        else if (clkVal && state == INPUT) {
          pins :> data;
          isa <: data;
          state = QUIET;
        }
        else if (clkVal && state == PENDQUIET) {
          isa <: 0;
          state = QUIET;
        }
        break;
      case (state == QUIET) => isa :> state :
        switch(state) {
          case OUTPUT :
            isa :> data;
            break;
          case INPUT :
            break;
        }
        break;
} } }
```

The declaration

```
void clkTimeStrobe(port pins, port clk, chanend isa)
```

declares `clkTimeStrobe` to be a function that takes a raw data port on which to sample and drive data, a raw port on which to sample clock edges and a channel end for interfacing with a processor. The type `pwidth_t` is the same size as the port width.

The function performs the following I/O operations on raw ports:

- `pins <: data;`

  The data in variable `data` is driven on the pins immediately.

- `pins :> data;`

  The value on the pins is read immediately and assigned to the variable `data`.

- `clk when pinsneq(clkVal) :> clkVal;`

  The value on the clock pin is input when it becomes unequal to its present value.

The clocking component waits for both clock edges and for requests from the processor. After receiving an output request, on the next falling edge of the clock, the data is driven and an acknowledgement is communicated to the processor. After receiving an input request, on the next sampling edge of the clock, data is sampled and communicated to the processor.

The program below shows the use of the clocked port by a thread, assumed to be executed on a processor, that performs the same sequence of outputs as the example given in §4.1. (Note that on XCore devices, the ports are interfaced directly by the ISA, rather than using a channel interface, so this program is not in practice generated.)

```
port p, c;

int main(void) {
  chan isa;
  par {
    clkTimeStrobe(p, c, isa);  // implemented in hardware
    for (int i=0; i<5; i++) {  // implemented in software
      isa <: OUTPUT;
      isa <: i;
      isa :> int;
    }
  }
}
```

The `for` loop outputs data to the function `clkTimeStrobe`, which then drives this data on the next falling edge of its clock. On XS1 devices, all ports used for data are buffered (see §C.1), which means that the acknowledgement happens almost immediately.

Note that XC's I/O semantics do not require a clock to provide edges at regular intervals, or even to provide edges at all. Neither do the semantics specify the state in which a port is initialised. The I/O semantics are therefore captured entirely within the `for` loop, with implementation-defined initialisation appearing outside of the loop (see §C.2). I/O timing characteristics are implementation-defined.

B.2 Clocking, Timing and Strobing Component

The behaviour of timed operations are defined as follows:

- A timed output causes the port to wait until its counter equals the specified time and then behaves as a clocked output.
- A timestamped output causes the processor to wait until the output is driven and to then record the value of the port counter at this time.
- A timed input causes the processor to wait until the port counter equals the specified time in the future and then behaves as a clocked input.

If a ready-in strobe signal is used:

- The ready-in signal is sampled on the rising edge of the port's clock.
- Input edges occur on rising edges of the port's clock when the ready-in signal is high.
- Output edges occur on falling edges of the port's clock when the ready-in signal was sampled high on the previous rising edge.

If a ready-out strobe signal is used:

- The ready-out signal is usually driven low.
- Output data is driven for at least a single period of the port's clock, and the ready-out signal is driven for the first period only; whether or not the output data continues to be driven is implementation-defined.
- The ready-out signal is driven high on the falling edge of the port's clock prior to the next rising edge on which data is sampled.

The functions below define the clocking/timing/strobing component, which can be configured to use a ready-in and ready-out signal and is capable of performing timed and timestamped operations.

```
void getInReq(chanend c, int &isTimed, counter_t &time, int &isTS) {
  /* protocol for inputting an `input request' from a channel */
  c :> isTimed;    // get time      (control)
  if (isTimed)
    c :> time;     // get time      (data)
  c :> isTS;       // get timestamp (control)
}

void getOutReq(chanend c, pwidth_t &data, int &isTimed,
               counter_t &time, int &isTS) {
  /* protocol for inputting an `output request' from a channel */
  c :> data;       // get output    (data)
  c :> isTimed;    // get time      (control)
  if (isTimed)
    c :> time;     // get time      (data)
  c :> isTS;       // get timestamp (control)
}
```

```
void clkTimeStrobe(port pins, int isReadyIn, port rin, int isReadyOut,
                   port rout, port clk,
                   chanend ser, chanend clrIn, chanend clrOut) {
  pwidth_t data     = 0;
  int state         = QUIET;
  int clkVal        = 0;
  int rinVal        = 0;
  int isTimed       = 0;
  counter_t counter = 0;
  counter_t time    = 0;
  int isTS          = 0;
  while (1) {
    select {
      case (state == QUIET) => ser :> state :
        switch(state) {
          case OUTPUT :
            getOutReq(ser, data, isTimed, time, isTS);
            break;
          case INPUT :
            getInReq(ser, isTimed, time, isTS);
            break;
          }
        break;
      case clk when pinsneq(clkVal) :> clkVal :
        if (!clkVal) { /* falling edge */
          counter++;
          if (state == OUTPUT && (!isTimed || counter == time))
            state = R_OUT;
          else if (state == INPUT  && (!isTimed || counter == time))
            state = R_IN;
          if (state == R_OUT && (!isReadyIn || rinVal)) {
            pins <: data;
            if (isReadyOut) rout <: 1;
            state = PENDQUIET;
          }
          else if (state == R_IN && isReadyOut)
            rout <: 1;
          else if (isReadyOut)
            rout <: 0;
        }
        else { /* rising edge */
          if (isReadyIn)
            rin :> rinVal;
          if (state == PENDQUIET) {
            ser <: 0; if (isTS) ser <: counter;
            state = QUIET;
          }
          else if (state == R_IN && (!isReadyIn || rinVal)) {
            pins :> data;
            ser  <: data; if (isTS) ser <: counter;
            state = QUIET;
           }
        }
        break;
      case clrIn :> int x : clrOut <: x; state = QUIET; break;
} } }
```

B.3 Serialisation Component

The behaviour of inputting and outputting on a serialised port is defined as follows:

- An output of a w-bit value on p pins is driven over w/p output edges, least significant bits first.
- The time specified by a timed or timestamped output represents the time from which the first p bits of data are driven; the processor blocks until the last p bits are driven.
- An input of a w-bit value on p pins is sampled over w/p input edges, with earlier bits received inserted in the least significant bits of w.
- The time specified by a timed or timestamped input represents the time from which the first[1] p data bits are read from the pins.

The functions below define the serialiser component:

```
void putOutReq(chanend cts, pwidth_t data, int isTimed,
               counter_t time, int isTS) {
  /* protocol for outputting an `output request' to a channel */
  cts <: OUTPUT;
  cts <: data;
  if (isTimed) {
    cts <: 1;
    cts <: time;
  }
  else
    cts <: 0;
  cts <: isTS;
}

void putInReq(chanend cts, int isTimed, counter_t time, int isTS) {
  /* protocol for inputting an `output request' from a channel */
  cts <: INPUT;
  if (isTimed) {
    cts <: 1;
    cts <: time;
  }
  else
    cts <: 0;
  cts <: isTS;
}
```

[1] On XS1 devices, the time specified by a timed or timestamped input is the time from which the *last* bits are sampled (see §C.1.1). This requires the serialiser to be continually active (push model), rather than being activated by the buffering component (pull model). It is anticipated—but not guaranteed—that future generations of the XMOS architecture will support the above semantics.

```
void serialiser(int w, int p, chanend cts, chanend bufIn, chanend bufOut,
                chanend clrIn, chanend clrOut) {
  twidth_t data  = 0;
  counter_t time = 0;
  counter_t ts   = 0;
  int clr        = 0;
  int isTimed    = 0;
  int isTS       = 0;

  while (1) {
    select {
      case bufIn :> int op :
        switch (op) {
          case OUTPUT :
            getOutReq(bufIn, data, isTimed, time, isTS);
            for (int i=0; i<w/p; i++) {
              pwidth_t chunk = ((1 << p) - 1) & data;
              data >>= p;
              putOutReq(cts, chunk, (isTimed && i == 0), time, 1);
              cts :> int;                    // synchronise
              cts :> ts;                     // get ts  (data)
            }
            bufIn <: 0;                      // synchronise
            if (isTS) bufIn <: ts;           // put ts     (control+data)
            break;
          case INPUT :
            getInReq(bufIn, isTimed, time, isTS);
            clr = 0;
            for (int i=0; i<w/p && !clr; i++) {
              if (i == 0)
                putInReq(cts, isTimed, time, isTS);
              else
                putInReq(cts, 0, time, 0);
              select {
                case cts :> pwidth_t chunk :  // get input  (data)
                  data = (data >> p) | (chunk << (w-p));
                  if (i == 0 && isTS)
                    cts :> ts;               // get ts     (data)
                  break;
                case clrIn :> int x :        // clear      (control)
                  clr = 1;
                  clrOut <: x;
                  break;
            } }
            if (!clr) {
              bufOut <: data & ((1<<w)-1);   // put output (data)
              if (isTS) bufOut <: ts;        // put ts     (data)
            }
            break;
        }
        break;
      case clrIn :> int x :
        clrOut <: x;
        break;
} } }
```

B.4 Buffering Component

A buffered port with a FIFO size of 0 is defined in the same way as if the buffering component is not present. The program below defines this "pass-through" behaviour, avoiding the need to remove the component from the model when buffering is not required.

```
void buffer(int size, int direction, chanend c, chanend, d, chanend isa,
            int isRout, chanend clrIn, chanend clrOut) {
  twidth_t  data    = 0;
  int       isTimed = 0;
  counter_t time    = 0;
  int       isTS    = 0;
  int       op      = 0;

  if (size == 0) {
    while (1) {
      isa :> int op;
      switch (op) {
        case OUTPUT :
          getOutReq(isa, data, isTimed, time, isTS);
          putOutReq(ser, data, isTimed, time, isTS);
          ser :> int;
          if (isTS) ser :> ts;
          isa <: 0;
          if (isTS) isa <: ts;
          break;
        case INPUT :
          getInReq(isa, isTimed, time, isTS);
          putInReq(ser, isTimed, time, isTS);
          ser :> data;
          isa <: data;
          if (isTS) {
            ser :> ts;
            isa <: ts;
       }  }
  } }
  else if (direction == OUT)
    portBufferOut(size, d, isa);
  else
    portBufferIn(size, d, e, isa, isRout, clrIn, clrOut);
}
```

B.4.1 FIFO Functions

The buffering components defined in the following sections make use of a standard first-in-first-out (FIFO) data structure, which is interfaced using the following functions:

`void addTail(FIFO f, twidth_t data, counter_t ts)`
: Adds an entry to the tail of the FIFO. If the FIFO is full, the oldest entry is removed to make room.

`{twidth_t, counter_t} getHead(FIFO f)`
: Returns the oldest entry from the head of the FIFO and removes it from the queue. If the FIFO is empty, the result returned is undefined.

`int isEmpty(FIFO f)`
: Returns non-zero if the FIFO is empty, and zero otherwise.

`int isFull(FIFO f)`
: Returns non-zero if the FIFO is full, and zero otherwise.

B.4.2 Buffered Output

The behaviour of outputting on a buffered port is defined as follows:

- An output inserts data into the port's FIFO, which performs the output once all pending outputs have completed; the processor blocks until the FIFO has space to accept the data.
- A timed output causes the processor to wait until the specified time and then performs the output.

The function below defines the buffering output component:

```
void portBufferOut(chanend ser, chanend isa) {
  FIFO b             = EMPTY;
  twidth_t  data     = 0;
  int       isTimed  = 0;
  counter_t time     = 0;
  int       isTS     = 0;
  counter_t ts       = 0;
  int       wait     = 0;

  while (1) {
    if (!isEmpty(b) && !wait) {
      {data, time} = getHead(b);
      putOutReq(ser, data, (time != NOTIME), time, isTS);
      wait = 1;
    }
    select {
      case !isFull(b) => isa :> int op :
        switch (op) {
          case OUTPUT :
            getOutReq(isa, data, isTimed, time, isTS);
            isa <: 0;
            if (isTimed)
              addTail(b, data, time);
            else
              addTail(b, data, NOTIME);
            break;
          case INPUT :
            /* implementation-defined behaviour */
            break;
         }
         break;
      case ser :> ts :
        if (isEmpty(b) && isTS)
          isa <: ts;
        wait = 0;
        break;
} } }
```

B.4.3 Buffered Input

The behaviour of inputting on a buffered port is defined as follows:

- On each input edge, data is sampled by the port and inserted into the port's FIFO; if the FIFO is full then the oldest value is discarded to make room for the most recently sampled value.
- An input fetches the next data from the FIFO; the processor blocks until the FIFO contains data.
- If a buffered port is configured with a ready-out strobe, the ready-out signal is driven high on each falling edge of the clock when the FIFO is not full.
- The time in a timed input represents time in the future; it causes the processor to discard any data in the buffer prior to performing the input.

The function below defines the buffering input component:

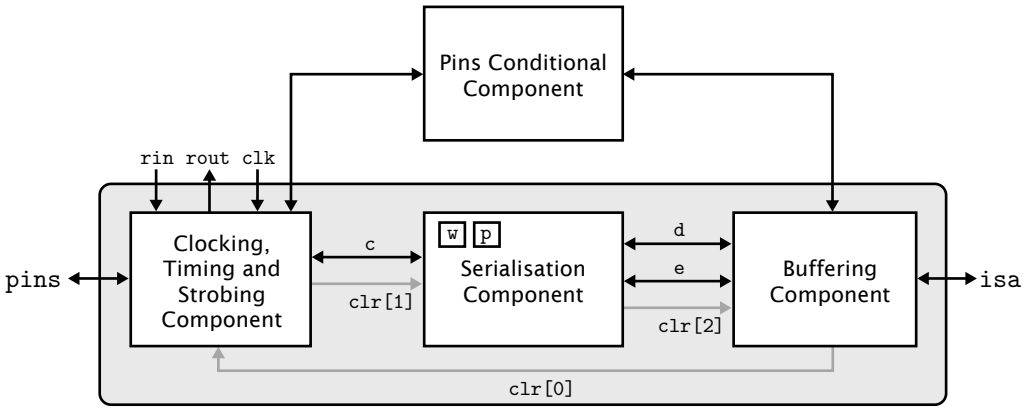
```
void portBufferIn(chanend serIn, chanend serOut, chanend isa,
                  int isReadyOut, chanend clrIn, chanend clrOut) {
  FIFO b         = EMPTY;
  twidth_t data  = 0;
  int isTimed    = 0, isTS = 0;
  counter_t time = 0, ts   = 0;
  while (1) {
    if (!isReadyOut || !isFull(b))
      putInReq(serIn, 0, 0, 1);
    select {
      case !isEmpty(b) => isa :> int op :
        switch (op) {
          case OUTPUT : /* implementation-defined behaviour */
            break;
          case INPUT :
            getInReq(isa, isTimed, time, isTS);
            if (!isTimed)
              {data, ts} = getHead(b);     // get buffered input (data)
            else {
              clrOut <: 1;                 // clear any pending inputs
              select {
                case clrIn :> int :
                  doEmpty(b);
                  break;
                case serOut :> twidth_t :
                  clrIn :> int; doEmpty(b);
                  break;
              }
              putInReq(serIn, 1, time, isTS);
              serOut :> data;              // get input  (data)
              serOut :> ts;                // get ts     (data)
            }
            isa <: data;                   // put input  (data)
            if (isTS) isa <: ts;           // put ts     (data)
            break;
        }
        break;
      case serOut :> data :                // get input  (data)
        serOut :> ts;                      // get ts     (data)
        addTail(b, data, ts);              // add buffer (data)
        break;
} } }
```

B.5 Conditional Input: pinseq and pinsneq

The behaviour of a conditional input on a clocked port is defined by predicate functions. The predicate functions pinseq and pinsneq are defined as follows:

- A conditional input with the function pinseq causes data to be sampled by the port on input edges until the value of the sampled (port-width) bits is equal to the specified paramter value; the processor waits until this time, taking the most recent data sampled.
- A conditional input with the function pinsnseq causes data to be sampled by the port on input edges until the value of the sampled (port-width) bits is unequal to the specified paramter value; the processor waits until this time, taking the most recent data sampled.
- A timed conditional input causes the processor to wait until the port counter equals the specified time and then behaves as a conditional input.

Semantically, both of these functions are defined as part of a *pins-conditional* component that appears between the clocking/timing/strobing component and the serialiser component.



This component requires the protocol used to communicate input requests through the buffering and serialisation components to be extended to pass a condition, which may be either *none*, *pins equal to* or *pins unequal to*. The function on the next page defines the pins-conditional component. For this component to be integrated:

- The clocking/timing/strobing component (see §B.2) is modified so that it can accept and respond to requests over an additional channel.
- The buffering input component (see §B.4) is modified to accept conditional input requests. If a conditional input is requested, the buffering component clears any pending inputs (as with timed inputs) and then communicates with this component over another channel.

```
void pinsConditional(chanend cts, chanend buf) {
  pwidth_t data     = 0;
  counter_t time    = 0;
  counter_t ts      = 0;
  int isTimed       = 0;
  int isTS          = 0;
  int n             = 0;
  int cond          = 0;
  pwidth_t condData = 0;
  int matched       = 0;

  while (1) {
    buf :> int;                      // get input req (control)
    getInReq(buf, isTimed, time, isTS);
    buf :> cond;
    buf :> condData;                 // get cond   (data)
    n = 0;
    matched = 0;
    while (!matched) {
      putInReq(cts, (isTimed && n == 0), time, isTS);
      cts :> data;                   // get input  (data)
      if (isTS)
        cts :> ts;                   // get ts     (data)
      switch (cond) {
        case NONE :
          matched = 1;
          break;
        case PINSEQ :
          matched = (data == condData);
          break;
        case PINSNEQ :
          matched = (data != condData);
          break;
    } }
    buf <: data;                     // snd data    (data)
    if (isTS)
      buf <: ts;                     // snd ts      (data)
    break;
  }
}
```

APPENDIX C

XS1 Implementation of XC

The following sections describe the XS1 implementation of XC, including the extent to which the I/O specification is implemented, the standard port library, port-to-pin mappings, and the size and alignment of types.

C.1 Support for XC Port Specification

The XC port declaration

```
port p;
```

declares a raw port. On XS1 devices, all ports used for inputting and outputting data are clocked by a 100MHz reference clock (see §C.2.1) and use a single-entry buffer, even if their declarations are not qualified with the keyword `buffered`.

The table below can be used to determine which I/O operations are supported on XS1 ports, depending on whether or not the corresponding XC declaration is qualified with the keyword `buffered`.

	Operation		
Mode	Serialisation	Strobing	`@ when`
Unqualified	✗	✗	✗
`buffered`	✓	✓	✓

A compiler is required to detect and issue an error in the following cases:

- Serialisation: A port not qualified with `buffered` is declared with a transfer width different from the port width.
- Strobing: A port not qualified with `buffered` is configured to use a ready-in or ready-out signal.
- An input uses both `@` and `when`: Both of these operators are used in an input statement with a port whose declaration is not qualified with `buffered`.

C.1.1 Serialisation

If serialisation is used (see §B.3), the time specified by a timed input statement records the time at which the *last* bits if data are sampled. This can result in unexpected behaviour when serialisation is used, since the construction

```
par {
  p @ t <: x;
  q @ t :> y;
}
```

causes the output on `p` to start at the same time as the input on `q` completes. To input and output this data in parallel, the input time should be offset in the software by an amount equal to the the transfer width divided by the port width.

C.1.2 Timestamping

The timestamp recorded by an input statement may come after the time when the data was sampled. This is because the XS1 provides separate instructions for inputting data and inputting the timestamp, so the timestamp can be input after the next data is sampled. This issue also affects output statements, but does not affect inputs performed in the guards of a `select` statement. A compiler should input the timestamp immediately after executing an input or output instruction, so in practice this behaviour is rarely seen.

C.1.3 Implementation-Defined Behaviour

An attempt to change the direction of a port qualified with `buffered` results in undefined behaviour.

C.2 XS1 Port Library: <xs1.h>

The header file `<xs1.h>` declares functions for configuring the mode of operation of ports and for operating on them.

C.2.1 Clock Configuration

An XS1 device provides a single reference clock that ticks at a frequency derived from an external oscillior. XC requires the system designer to ensure that the reference clock ticks at 100MHz for correct operation of timers.

Each XCore provides a set of programmable *clock blocks*, which can be used to produce clock signals for ports. A clock block can use either a 1-bit port or a divided reference clock. The header file `<xs1.h>` provides a resource type `clock`. A variable of type `clock` must be declared globally and initialised with a unique clock block resource identifier, as in:

```
clock c = XS1_CLKBLK_1;
```

The number of clock blocks available is given in the device datasheet. Their names are as the above declaration, numbered sequentially from 1.

The functions below are used to configure a clock block.

`void configure_clock_src(clock clk, void port p)`
: `configure_clock_src` configures a clock to use a 1-bit port as its source. If the port is not 1-bit wide, an exception is raised.

`void configure_port_clock_output(void port p, const clock clk)`
: `configure_port_clock_output` configures a 1-bit port to drive a clock signal. If the port is not 1-bit wide, an exception is raised. If the clock is internally-generated and has a rate of 100MHz, no output is driven. Performing inputs or outputs on a port configured in this mode results in undefined behaviour.

`configure_clock_rate(clock clk, unsigned a, unsigned b)`
: This function configures a clock to run at a rate of $\frac{a}{b}$MHz. If the specified rate is not supported by the hardware, an exception is raised. The hardware supports a rate of 100MHz and rates of the form $(50/n)$MHz where n is in the range 1 to 255 inclusive. A 100MHz reference clock is required for correct operation.

`void configure_clock_rate_at_most(clock clk, unsigned a, unsigned b)`
`void configure_clock_at_least(clock clk, unsigned char divide)`
: These functions configure a clock to run at the fastest/slowest non-zero rate supported by the hardware that is less than or equal to $\frac{a}{b}$ MHz. An exception is raised if no rate satisfies this criterion. A 100MHz reference clock is required for correct operation.

`void set_clock_fall_delay(clock clk, unsigned n)`
`void set_clock_rise_delay(clock clk, unsigned n)`
: These functions cause the falling/rising edge of the clock to be delayed by n processor-clock cycles before it is seen by any port connected to the clock. The delay must be a value in the range 0 to 512 inclusive. If the clock edge is delayed by more than the clock period, no falling/rising edges are seen by any port connected to the clock.

`void start_clock(clock clk)`
: `start_clock` causes the clock to start generating edges, and resets the counters for all of the ports connected to the clock to 0.

`void stop_clock(clock clk)`
: `stop_clock` causes the processor to wait until a clock is low and then puts it into a stopped state (in which it does not generate edges).

C.2.2 Port Configuration

The functions below are used to change the mode of operation of a port.

```
void configure_in_port(void port p, const clock clk)
void configure_in_port_strobed_master
       (void port p, out port readyout, const clock clk)
void configure_in_port_strobed_slave
       (void port p, in port readyin, clock clk)
```

```
void configure_in_port_handshake
       (void port p, in port readyin, out port readyout, clock clk)
```
These functions configure a port with a specified clock, ready-in and ready-out signals in input mode. If either the ready-in or ready-out ports are not 1-bit wide, an exception is raised. (See §B.2 and §C.1.)

```
void configure_out_port(void port p, const clock clk, unsigned initial)
void configure_out_port_strobed_master
       (void port p, out port readyout, const clock clk, unsigned initial)
void configure_out_port_strobed_slave
       (void port p, in port readyin, clock clk, unsigned initial)
void configure_out_port_handshake
  (void port p, in port readyin, out port readyout, clock clk, unsigned initial)
```
These functions configure a port with a specified clock, ready-in and ready-out signals in output mode, causing the initial value to be driven immediately. If either the ready-in or ready-out ports are not 1-bit wide, an exception is raised. (See §B.2 and §C.1.)

```
void set_pad_delay(void port p, unsigned n)
```
`set_pad_delay` sets a delay on the pins connected to a port. The input signals sampled on the port's pins are delayed by *n* processor-clock cycles before they are seen on the port. The hardware supports delay values from 0 to 5 inclusive. If multiple enabled ports are connected to the same pin (see §C.3), the delay on that pin is set to that of the highest priority port.

```
void set_port_inv(void port p)
void set_port_no_inv(void port p)
```
`set_port_inv` configures a 1-bit port to invert the data which is sampled and driven on its pins. If the port is used as the source for a clock, setting this mode has the effect of the swapping the rising and falling edges of the clock. `set_port_no_inv` ensures that the port does not invert the data. If the port is not a 1-bit port, an exception is raised.

```
void set_port_sample_delay(void port p)
void set_port_no_sample_delay(void port p)
```
These functions set the sampling edge of a port. The first function sets the sampling edge to the falling edge of the port's clock; second sets it to the rising edge. If the sample delay is set to the falling edge of the clock, a timed input with a ready-out signal causes the ready-out signal to be driven at the specified time, and the input to be completed a single period later.

```
void set_port_drive(void port p)
void set_port_pull_up(void port p)
```
These functions configure a port to be in either drive or pull-up mode (by default, a port is configured in drive mode, the behaviour of which is described in Appendix B). In pull-up mode, when the port is used for input, its internal pull-up resistor is enabled. If the port is not 1-bit wide, values driven on the pins are undefined. For 1-bit ports, an output of 0 causes the port to drive its pin low, and

an output of 1 causes no value to be driven; when the pin is not driving data, the pull-up resistor ensures that the value sampled by the port is high. The pull-up is not strong enough to guarantee a defined external value.

C.2.3 Input and Output Operations

`void pinseq(void port p, unsigned val)`
`void pinsneq(void port p, unsigned val)`
These functions request a conditional input from the port. (See §B.5.) The functions must be called as the `when` expression of an input on a port, omitting the port from the call.

`void timerafter(timer t, unsigned val)`
`timerafter` causes the procecssor to wait until the value of the timer's counter is interpreted as coming after the specified value. The time *A* is considered to come after the time *B* if `((int)(`*B* `-` *A*`) < 0)` is true. This function must be called as the `when` expression of an input on a timer, omitting the timer from the call.

`void partout(void port p, unsigned bits, unsigned val)`
`unsigned partout_timestamped(void port p, unsigned bits, unsigned val)`
`unsigned partout_timed(void port p, unsigned bits, unsigned val, unsigned t)`
These functions output the least significant *bits* bits of *val* to the specified port; the second function timestamps the output; the third function causes the output to be driven when the port counter equals the specified time. The port must be declared with the qualifier `buffered`. The number of bits must be less than the transfer width of the port, greater than zero and a multiple of the port width, otherwise an exception is raised.

`unsigned endin(void port p)`
`endin` causes the port to end the current input on a port. The port must be declared with the qualifier `buffered`. The number of bits sampled by the port but not yet input by the processor is returned, which includes any data in the FIFO or serialisation register. Subsequent inputs on the port cause the port to provide transfer-width bits of data until there is less than one transfer-width bits of data remaining. Any remaining data can be read with one further input, which provides transfer-width bits of data with the remaining buffered data in the most significant bits of this value.

`void sync(void port p)`
`sync` causes the processor to wait until a port has driven all pending outputs and the last port-width bits of data has been held on the pins for one clock period.

`void clearbuf(void port p)`
`clearbuf` causes the port to discard any data in its FIFO. If the port is currently driving data on its pins, the data continues to be driven; if the port is serialising output, the current data continues to be driven with any remaining data discarded.

`unsigned peek(void port p)`

`peek` causes the port to sample the current value on its pins. The port provides the sampled port-width bits of data to the processor immediately, regardless of its transfer width, clock, ready signals and buffering. The input has no effect on subsequent I/O performed on the port.

C.3 Specifying Port-to-Pin Mappings

On XS1 devices, pins are used to interface with external components via ports and to construct links to other devices over which channels are established. The ports are multiplexed, allowing the pins to be configured for use by ports of different widths. Table I gives the XS1 port-to-pin mapping, which is interpreted as follows:

- The name of each pin is given in the format XnDpq where n is a valid XCore number for the device and pq exists in the table. The physical position of the pin depends on the packaging and is given in the device datasheet.
- Each link is identified by a letter A-D. The wires of a link are identified by means of a superscripted digit 0-4.
- Each port is identified by its width (the first number 1, 4, 8, 16 or 32) and a letter that distinguishes multiple ports of the same width (A-P). These names correspond to port identifiers in the header file `<xs1.h>` (for example port 1A corresponds to the identifier `XS1_PORT_1A`). The individual bits of the port are identified by means of a superscripted digit 0-31.
- The table is divided into six rows (or *banks*). The first four banks provide a selection of 1, 4 and 8-bit ports, with the last two banks enabling the single 32-bit port. Different packaging options may export different numbers of banks; the 16-bit and 32-bit ports are not available on small devices.

The ports used by a program are determined by the set of XC port declarations. For example, the declaration

```
on stdcore[0] : in port p = XS1_PORT_1A;
```

uses the 1-bit port 1A on XCore 0, which is connected to pin X0D00.

Usually the designer should ensure that there is no overlap between the pins of the declared ports, but the precedence has been designed so that, if required, portions of the wider ports can be used when overlapping narrower ports are used. The ports to the left of the table have precedence over ports to the right. If two ports are declared that share the same pin, the narrower port takes priority. For example:

```
on stdcore[2] : out port p1 = XS1_PORT_32A;
on stdcore[2] : out port p2 = XS1_PORT_8B;
on stdcore[2] : out port p3 = XS1_PORT_4C;
```

In this example:

- I/O on port `p1` uses pins X2D02 to X2D09 and X2D49 to X2D70.
- I/O on port `p2` uses pins X2D16 to X2D19; inputting from `p2` results in undefined values in bits 0, 1, 6 and 7.
- I/O on port `p3` uses pins X2D14, X2D15, X2D20 and X2D21; inputting from `p1` results in undefined values in bits 28-31, and when outputting these bits are not driven.

TABLE I
AVAILABLE PORTS AND LINKS FOR EACH PIN

Pin	link	⇐ highest Precedence 1-bit ports	4-bit ports	8-bit ports	16-bit ports	lowest ⇒ 32-bit port
X*n*D00		$1A$				
X*n*D01	A^4 in/out	$1B$				
X*n*D02	A^3 in/out		$4A^0$	$8A^0$	$16A^0$	$32A^{20}$
X*n*D03	A^2 in/out		$4A^1$	$8A^1$	$16A^1$	$32A^{21}$
X*n*D04	A^1 in/out		$4B^0$	$8A^2$	$16A^2$	$32A^{22}$
X*n*D05	A^0 in/out		$4B^1$	$8A^3$	$16A^3$	$32A^{23}$
X*n*D06	A^0 out/in		$4B^2$	$8A^4$	$16A^4$	$32A^{24}$
X*n*D07	A^1 out/in		$4B^3$	$8A^5$	$16A^5$	$32A^{25}$
X*n*D08	A^2 out/in		$4A^2$	$8A^6$	$16A^6$	$32A^{26}$
X*n*D09	A^3 out/in		$4A^3$	$8A^7$	$16A^7$	$32A^{27}$
X*n*D10	A^4 out/in	$1C$				
X*n*D11		$1D$				
X*n*D12		$1E$				
X*n*D13	B^4 in/out	$1F$				
X*n*D14	B^3 in/out		$4C^0$	$8B^0$	$16A^8$	$32A^{28}$
X*n*D15	B^2 in/out		$4C^1$	$8B^1$	$16A^9$	$32A^{29}$
X*n*D16	B^1 in/out		$4D^0$	$8B^2$	$16A^{10}$	
X*n*D17	B^0 in/out		$4D^1$	$8B^3$	$16A^{11}$	
X*n*D18	B^0 out/in		$4D^2$	$8B^4$	$16A^{12}$	
X*n*D19	B^1 out/in		$4D^3$	$8B^5$	$16A^{13}$	
X*n*D20	B^2 out/in		$4C^2$	$8B^6$	$16A^{14}$	$32A^{30}$
X*n*D21	B^3 out/in		$4C^3$	$8B^7$	$16A^{15}$	$32A^{31}$
X*n*D22	B^4 out/in	$1G$				
X*n*D23		$1H$				
X*n*D24		$1I$				
X*n*D25		$1J$				
X*n*D26			$4E^0$	$8C^0$	$16B^0$	
X*n*D27			$4E^1$	$8C^1$	$16B^1$	
X*n*D28			$4F^0$	$8C^2$	$16B^2$	
X*n*D29			$4F^1$	$8C^3$	$16B^3$	
X*n*D30			$4F^2$	$8C^4$	$16B^4$	
X*n*D31			$4F^3$	$8C^5$	$16B^5$	
X*n*D32			$4E^2$	$8C^6$	$16B^6$	
X*n*D33			$4E^3$	$8C^7$	$16B^7$	
X*n*D34		$1K$				
X*n*D35		$1L$				
X*n*D36		$1M$		$8D^0$	$16B^8$	
X*n*D37		$1N$		$8D^1$	$16B^9$	
X*n*D38		$1O$		$8D^2$	$16B^{10}$	
X*n*D39		$1P$		$8D^3$	$16B^{11}$	
X*n*D40				$8D^4$	$16B^{12}$	
X*n*D41				$8D^5$	$16B^{13}$	
X*n*D42				$8D^6$	$16B^{14}$	
X*n*D43				$8D^7$	$16B^{15}$	
X*n*D49	C^4 in/out					$32A^0$
X*n*D50	C^3 in/out					$32A^1$
X*n*D51	C^2 in/out					$32A^2$
X*n*D52	C^1 in/out					$32A^3$
X*n*D53	C^0 in/out					$32A^4$
X*n*D54	C^0 out/in					$32A^5$
X*n*D55	C^1 out/in					$32A^6$
X*n*D56	C^2 out/in					$32A^7$
X*n*D57	C^3 out/in					$32A^8$
X*n*D58	C^4 out/in					$32A^9$
X*n*D61	D^4 in/out					$32A^{10}$
X*n*D62	D^3 in/out					$32A^{11}$
X*n*D63	D^2 in/out					$32A^{12}$
X*n*D64	D^1 in/out					$32A^{13}$
X*n*D65	D^0 in/out					$32A^{14}$
X*n*D66	D^0 out/in					$32A^{15}$
X*n*D67	D^1 out/in					$32A^{16}$
X*n*D68	D^2 out/in					$32A^{17}$
X*n*D69	D^3 out/in					$32A^{18}$
X*n*D70	D^4 out/in					$32A^{19}$

C.4 Channel Communication

On some revisions of the XS1 architecture, it is not possible to input data of size less than 32 bits from a streaming channel in the guard of a `select` statement.

C.5 Data Types

The size and alignment of XC's data types are not specified by the language. This allows the size of `int` to be set to the natural word size of the target device, ensuring the fastest possible performance for many computations. It also allows the alignment to be set wide enough to enable efficient memory loads and stores. Table II gives the size and alignment of the data types specified by the XMOS Application Binary Interface [8], which provides a standard interface for linking objects compiled from both XC and C. In addition:

- The `char` type is by default unsigned.
- The types `char`, `short` and `int` may be specified in a bit-field's declaration.
- `enum` bit-fields are unsigned unless the enum has negative values.
- A zero-width bit-field forces padding until the next bit-offset aligned with the bit-field's declared type.
- The notional transfer type of a port is `unsigned int` (32 bits).
- The notional counter type of a port is `unsigned short` (16 bits).
- The notional counter type of a timer is `unsigned int` (32 bits).

TABLE II
SIZE AND ALIGNMENT OF DATA TYPES ON XS1 DEVICES

Data Type	Size (bits)	Align (bits)	Supported XC	Supported C	Meaning
`char`	8	8	✓	✓	Character type
`short`	16	16	✓	✓	Short integer
`int`	32	32	✓	✓	Native integer
`long`	32	32	✗	✓	Long integer
`long long`	64	32	✗	✓	Long long integer
`float`	32	32	✗	✓	32-bit IEEE float
`double`	64	32	✗	✓	64-bit IEEE float
`long double`	64	32	✗	✓	64-bit IEEE float
`void *`	32	32	✗	✓	Data pointer
`port`	32	32	✓	✗	Port
`timer`	32	32	✓	✗	Timer
`chanend`	32	32	✓	✗	Channel end

Bibliography

[1] David May. *The XMOS XS1 Architecture*. XMOS Limited, 2009.

[2] Douglas Watt and Huw Geddes. *The XMOS Tools User Guide*. XMOS Limited, 2009.

[3] Peter Hedinger and Ali Dixon and Ross Owen and Neil Richards and David May and Henk Muller. XS1 Ports: use and specification. Website, 2008. `http://www.xmos.com/published/xs1-portspec`.

[4] Hitachi TX14 Series LCD. Website, 2008. `http://www.farrell.com/datasheets/71533.pdf`.

[5] IEEE 802.3 Section 2. Website, 2008. `http://standards.ieee.org/getieee802/download/802.3-2005_section2.pdf`.

[6] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1988.

[7] International Organization for Standardization. *ISO/IEC 9899:1999: Programming Languages — C*. Wiley, West Sussex, England, December 1999.

[8] Douglas Watt and Richard Osborne and Martin Young. XMOS XS1 32-Bit Application Binary Interface. Website, 2009. `http://www.xmos.com/published/abi97`.

Index

A

Q

R

S

V

W

X